Strongly Typed and Efficient Functional Reactive Programming

Von der Fakultät für Mathematik, Naturwissenschaften und Informatik
der Brandenburgischen Technischen Universität Cottbus

zur Erlangung des akademischen Grades

Doktor der Naturwissenschaften
(Dr. rer. nat.)

genehmigte Dissertation

vorgelegt von

Diplom-Informatiker

Wolfgang Jeltsch

geboren am 4. November 1976 in Herzberg/Elster

Gutachter: Prof. Dr. rer. nat. habil. Petra Hofstedt
Gutachter: Prof. Dr. rer. nat. habil. Peter Bachmann
Gutachter: Prof. Dr. rer. nat. habil. Peter Pepper

Tag der mündlichen Prüfung: 8. Dezember 2011

Contents

List of Figures

1. Introduction

Functional programming is valued for its high level semantics and its extensive support for modularity, which lead to an increase in programmer productivity, maintainability, and reliability. While functional programming is successfully used in a variety of fields nowadays, there are areas where it has not gained much ground yet. One such area are systems where temporal aspects and reactivity play a major role. For example, GUI programming is mostly carried out in an imperative style today, even when using a functional programming language such as Haskell. This would not be a problem if imperative programming would be a natural paradigm for dealing with reactive systems. And in fact, there are even functional programmers who think it is. However, we disagree with this view. To see why, let us take a closer look at imperative GUI programming.

An ordinary imperative program directly specifies the complete control flow of the application. However, an imperative GUI program is mainly a collection of event handlers – code fragments that tell how to react to single events. The program does not directly describe what events trigger what actions. Instead, it describes technical details for realizing reactivity. Event handlers are registered in order to be called when certain events occur, and a global event loop waits for events and dispatches them to the corresponding handlers.

This low-level nature of imperative GUI programming can quickly lead to subtle problems. For example when the state of the system changes in reaction to an event, inconsistent intermediate states become visible to the program. So operations that expect a consistent state might misbehave. Another problem of low-level GUI programming are infinite event cascades, caused by circular dependencies between GUI components.

The imperative approach to reactive systems is also rather low-level when it comes to continuous change. Say we want to include a graphical animation into a GUI application. Then the program has to explicitely deal with the fact that the desired behavior of the animation can only be approximated by discrete sequences of frames.

Let us now look away from the technical details to get a more abstract view on reactive systems. Ideally, we want events to change the system state immediately, without time consumption and without causing inconsistent intermediate states. Furthermore, we want to be able to have truly continuous change. We know that a computer cannot realize such idealized behavior, but we want a programming model that is able to express it.

Such a programming paradigm has already been invented. In 1997, Conal Elliott and Paul Hudak [8] came up with a Haskell library called “Fran”, meaning

"Functional Reactive Animation". In Fran, events and continuously changing values are first class objects. By using a set of combinator functions, complex descriptions of temporal behavior can be constructed.

While Fran dealt only with animations, its general approach to temporal descriptions proved to be valuable for other application areas too. So the universal concepts were factored out of Fran and named Functional Reactive Programming (FRP) [32]. A number of different FRP systems have been developed since then. Among them are Haskell libraries like FranTk [25], Yampa [22, 10], and Reactive [7] as well as implementations in other languages, most notably, Frappé [5] (Java), FrTime [4] (Scheme), and Flapjax [21] (JavaScript).

One particular reason for the high number of FRP systems is the ongoing quest for efficient FRP implementations that respect simple and intuitive semantics. It has turned out that finding such an implementation is a non-trivial task. Our desire to improve the state of the art in this field led to the development of Grapefruit[1], another Haskell FRP library. The novel concepts we describe in this thesis came out of our work on Grapefruit.

In the next chapter, we give the reader a taste of FRP by means of an example. Chapter 3 defines an FRP interface and an accompanying semantics, which are used throughout this thesis. Afterwards, we make the following contributions:

- In Chapter 4, we discuss important approaches to implementing FRP that were developed by other authors. The FRP systems that are based on these approaches often use different interfaces, semantics, and terminology. However, we discuss all the implementation ideas in the context of our single FRP specification from Chapter 3, thus making comparison of the different approaches simpler. To our knowledge, this is the first survey of this kind.

- Chapter 5 shows how start times of temporal descriptions can be represented by type parameters. This allows us to get rid of issues with performance and semantics that occur in several FRP systems.

- In Chapter 6, we present a new implementation of the discrete FRP subset. This implementation is simple, efficient, and semantically correct at the same time.

- FRP profits from a record system that allows for the definition of generic record combinators while retaining static type checks. Chapter 7 presents such a system, which is implemented as a Haskell library. This record system is not tied to FRP, but can be used in other areas too.

- Chapter 8 shows how to emulate subkinds – the kind-level analogon to subtypes – in Haskell. While this is an interesting development in itself, we use it mainly for making the record system from Chapter 7 more general.

We give conclusions and discuss further work in Chapter 9.

[1] See `http://grapefruit-project.org/`.

2. FRP by Example

As an example, we create a small network monitoring application. This application tracks all incoming and outgoing network packets and displays the total volume of either incoming traffic, outgoing traffic, or all traffic.

Network traffic consists of a sequence of packets, each sent or received at a specific time. A pair of a time and an associated value is called an event in FRP, and a sequence of events is a discrete signal. So we can model all future incoming traffic as a single discrete signal, and all future outgoing traffic as another one. Given a type *Packet* of all possible packets, these discrete signals are values of type *DSignal Packet*.

Let us assume that we have two I/O actions *getInTraffic* and *getOutTraffic* of type *IO* (*DSignal Packet*) that yield the discrete signals of sent and received packets. We start our code as follows:

$$
\begin{aligned}
&\mathit{monitor} :: \mathit{IO}\ () \\
&\mathit{monitor} = \mathbf{do} \\
&\qquad \ddot{p}_{In}\ \ \leftarrow \mathit{getInTraffic} \\
&\qquad \ddot{p}_{Out} \leftarrow \mathit{getOutTraffic} \\
&\qquad [\ldots]
\end{aligned}
$$

We use identifiers with two dots on top for discrete signals. The dots illustrate the discrete events the signals consist of. Since we use the identifier p for single packets, we use $\ddot{p}$ for discrete signals of packets.

Traffic volume, measured in bytes, is an integer that varies over time. FRP provides us with the concept of continuous signal for modelling time-varying values. We could represent traffic volume by a value of type *CSignal Integer*. However, traffic volume only changes when a packet is sent or received. In such a situation, it is better to use a segmented signal. Roughly speaking, a segmented signal is a continuous signal that changes its value only at discrete points in time. A segmented signal of integers is a value of type *SSignal Integer*. We use identifiers with a tilde ($\tilde{}$) for continuous signals and identifiers with a bar ($\bar{}$) for segmented signals.

Let us now turn the discrete signals $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ into segmented signals that describe the time-varying volume of incoming and outcoming traffic, respectively. We add the following two declarations for this purpose:

$$
\begin{aligned}
\bar{v}_{In}\ &= \mathit{volume}\ \ddot{p}_{In} \\
\bar{v}_{Out} &= \mathit{volume}\ \ddot{p}_{Out}
\end{aligned}
$$

The function *volume*, which is used in these declarations, is defined as follows:

$$volume :: DSignal\ Packet \to SSignal\ Integer$$
$$volume = scanl\ (\lambda v\ p \to v + size\ p)\ 0$$

This definition requires a function *size* of type $Packet \to Integer$ that calculates the size of a given packet. The combinator *scanl*, which has type $(\beta \to \alpha \to \beta) \to \beta \to DSignal\ \alpha \to SSignal\ \beta$, is similar to the *scanl* from Haskell's *Data.List* module. A segmented signal $scanl\ f\ y_0\ \ddot{x}$ starts with the value y_0. At every event in $\ddot{x}$ with value x, it changes from its current value y to the value $f\ y\ x$.

Now, we want to calculate the total traffic volume, covering both incoming and outgoing traffic. We can do this in two ways. The first way is to construct a discrete signal that represents all network traffic, and apply the *volume* function to it. The signal of all network traffic is the union of $\ddot{p}_{In}$ and $\ddot{p}_{Out}$, written $\ddot{p}_{In}$ `` `union` `` $\ddot{p}_{Out}$.

Usually, the union of two discrete signals $\ddot{x}_1$ and $\ddot{x}_2$ covers all events from $\ddot{x}_1$ and $\ddot{x}_2$. There is, however, one exception. If at some time both $\ddot{x}_1$ and $\ddot{x}_2$ have an event, the event from $\ddot{x}_2$ is not included into the union. This is to avoid that a single discrete signal covers two events that occur at the same time. In our example, this exception does not come into effect, since we can assume that there will never be a packet sent at the same time another packet is received.

We do not need this assumption if we use the second method of calculating the total traffic volume. *SSignal* is an applicative functor [20]. For each n-ary function f, the function $liftA_n\ f$ turns segmented signals $\bar{x}_1$ through $\bar{x}_n$ into a segmented signal $\bar{y}$ such that $\bar{y}$ has the value $f\ x_1\ \ldots\ x_n$ at a time t if every $\bar{x}_i$ has the value x_i at time t. So $liftA_2\ (+)\ \bar{v}_{In}\ \bar{v}_{Out}$ denotes the total traffic volume, which we call $\bar{v}_{All}$.

The user shall be able to select whether the volume of incoming traffic, outgoing traffic, or all traffic is shown. We introduce a type *KindOfTraffic*, whose values denote these different kinds of traffic:

$$\textbf{data}\ KindOfTraffic = In \mid Out \mid All$$

We assume that we have an I/O action *makeKindOfTrafficSelector*, which has type *IO* (*SSignal KindOfTraffic*). Executing *makeKindOfTrafficSelector* creates a GUI widget that allows the user to choose between the three kinds of traffic, and returns a segmented signal $\bar{k}$ that contains the current selection for each time.

Now, we want to build a segmented signal $\bar{v}$ that always mirrors the currently selected traffic volume. To do so, we first construct a "higher-order signal" $\bar{\bar{v}}$ of type *SSignal* (*SSignal Integer*) whose value is $\bar{v}_k$ whenever $\bar{k}$ has the value k. We can define $\bar{\bar{v}}$ as follows:

$$\bar{\bar{v}} = fmap\ (\lambda k \to \textbf{case}\ k\ \textbf{of}\ In \to \bar{v}_{In}; Out \to \bar{v}_{Out}; All \to \bar{v}_{All})\ \bar{k}$$

Afterwards, we turn $\bar{\bar{v}}$ into $\bar{v}$ by applying the *switch* combinator to it. The function *switch* transforms a segmented signal $\bar{s}$ of signals into an ordinary signal s. Everytime $\bar{s}$ changes its value to a new signal, s starts to behave like that signal.

Finally, we want to display the traffic volumes that are provided by $\bar{v}$. We assume that there is a function *makeStringDisplay* of type $SSignal\ String \to IO\ ()$.

```
monitor :: IO ()
monitor = do
            p̈In  ← getInTraffic
            p̈Out ← getOutTraffic
            let
              v̄In  = volume p̈In
              v̄Out = volume p̈Out
              v̄All = liftA2 (+) v̄In v̄Out
            k̄ ← makeKindOfTrafficSelector
            let
              v̿ = fmap (λk → case k of In → v̄In; Out → v̄Out; All → v̄All) k̄
              v̄ = switch v̿
            makeStringDisplay (fmap show v̄)
            run
volume :: DSignal Packet → SSignal Integer
volume = scanl (λv p → v + size p) 0
```

Figure 2.1.: Source code of the network monitor

```
scanl  :: (β → α → β) → β → DSignal α → SSignal β
fmap   :: (α → β) → SSignal α → SSignal β
liftA2 :: (α → β → γ) → SSignal α → SSignal β → SSignal γ
switch :: SSignal (SSignal α) → SSignal α
run    :: IO ()
```

Figure 2.2.: FRP core support used by the network monitor

Applying *makeStringDisplay* to a signal $\bar{w}$ creates a GUI widget that always shows the current value of $\bar{w}$. We can construct a display for the selected traffic volumes with the I/O action *makeStringDisplay* (*fmap show* $\bar{v}$).

Having set up our application, we can now run it by executing the predefined I/O action *run*. The application now runs autonomously. The FRP library takes care to update the traffic volume display when needed, that is, when the user chooses a different kind of traffic for display, or when a monitored packet passes a network interface.

Figure 2.1 shows the complete source code of the network monitor. An overview of the external types and values used by the network monitor is given in Figure 2.2 and Figure 2.3.

```
data Packet          = [...]              -- implementation not relevant here
data KindOfTraffic = In | Out | All
size                          :: Packet → Integer
getInTraffic                  :: IO (DSignal Packet)
getOutTraffic                 :: IO (DSignal Packet)
makeKindOfTrafficSelector :: IO (SSignal KindOfTraffic)
makeStringDisplay             :: SSignal String → IO ()
```

Figure 2.3.: Application-oriented support used by the network monitor

3. Interface and Semantics

FRP is about working with signals, which describe temporal phenomena. There are three different kinds of signals – discrete, continuous, and segmented. These correspond to the type constructors *DSignal*, *CSignal*, and *SSignal*, which we already saw in Chapter 2. Section 3.1 presents a formal semantics for these type constructors. FRP also covers a variety of signal combinators. These combinators, including their semantics, are discussed in Section 3.2. Several FRP implementations do not support signals directly, but only so-called generators. We talk about generators in Section 3.3.

For describing semantics, we use Haskell instead of common mathematical notation. This gives us a richer expression syntax and access to a variety of predefined types and operators. Furthermore, it allows us to restrict our semantics definition to the actual FRP aspects. Wherever the semantics has to deal with ordinary types and values, it uses these types and values directly. So there is no need to explicitely define meanings of constructs that are not unique to FRP.

The meaning of a signal type constructor S is a unary type synonym $[\![S]\!]$. The values of a type $[\![S]\!]\ \alpha$ are the meanings of those signals that have type $S\ \alpha$. We use identifiers with two dots ($\ddot{}$), a tilde ($\tilde{}$), and a bar ($\bar{}$) not only for signals, but also for signal meanings. Each signal combinator f has a meaning $[\![f]\!]$, which performs the same operation as f, but works with signal meanings instead of signals. As a consequence, the type of $[\![f]\!]$ can be derived from the type of f by replacing every occurrence of a signal constructor S with its meaning $[\![S]\!]$.

To do anything useful, we need to be able to interact with the real world. We use specific I/O actions for this, which we call producers and consumers. A producer generates a signal that mirrors a part of the real world, and a consumer takes one or more signals and influences the real world based on them. The actions *getInTraffic*, *getOutTraffic*, and *makeKindOfTrafficSelector* from Chapter 2 are examples of producers, while *makeStringDisplay* is an example of a consumer. Finally, there is an I/O action *run* that actually executes the reactive system. We do not discuss producers, consumers, and the *run* action further in this section.

Note that different FRP systems often differ slightly in their underlying semantics. For example, some systems allow discrete signals to contain multiple events at the same time, which we disallow. In other systems, a continuous signal has a value also at the program start, while in our semantics, continuous signals only really start immediately after the program's start time. However, such differences do not become relevant when we refer to other FRP systems. So we use the semantics outlined below consistently throughout this thesis.

Different FRP systems also use different terminology sometimes. For example,

discrete signals are often called event streams or simply events. Furthermore, type constructors and combinators are often named differently. Again, we value consistency highly, so that we use our terms and identifiers even when referring to other people's work.

3.1. Signal Types

This section introduces the semantics of the signal type constructors *DSignal*, *CSignal*, and *SSignal* in Subsections 3.1.1, 3.1.3, and 3.1.4, respectively. A necessity for a formal signal semantics is a precise notion of time. We will discuss this issue in Subsection 3.1.2.

3.1.1. Discrete Signals

Let us first develop the semantics of discrete signals. A discrete signal is a sequence of events. An event occurs at a specific time and carries a value that may give additional information about the event. So it seems natural to use lists of time–value pairs as meanings of discrete signals, which leads to the following definition of $[\![DSignal]\!]$:

$$\textbf{type } [\![DSignal]\!]\ \alpha = [(Time, \alpha)]$$

Here, the type *Time* shall be the type of all points in time.

While being simple, this definition is too permissive. We want the order of the time–value pairs to correspond to the temporal order of the events they describe. Furthermore, we want to avoid that a discrete signal covers two events that occur at the same time. So we want the times in a discrete signal meaning to be strictly increasing. Alas, this extra requirement cannot be formulated using Haskell types.

Fortunately, we can modify the discrete signal semantics to achieve the same effect with pure Haskell. We replace the absolute times by positive time differences. Assuming a type *PTD* of all positive time differences, we define $[\![DSignal]\!]$ as follows:

$$\textbf{type } [\![DSignal]\!]\ \alpha = [(PTD, \alpha)]$$

Let t_0 be the time when the program started. A meaning $[(\Delta t_1, x_1), (\Delta t_2, x_2), \ldots]$ denotes a discrete signal that has an event with value x_i at time $t_0 + (\Delta t_1 + \ldots + \Delta t_i)$ for every applicable i.

An additional restriction of this new semantics is that a discrete signal cannot cover events that occur at the program start or before. However, this is not a problem. Usually, a program does not know about events that occurred before it started anyway. Therefore, we just exclude them from discrete signals. Events that occur at the program start can be handled with an extended discrete signal type that is derived from *DSignal* as follows:

$$\textbf{data } ExtDSignal\ \alpha = ExtDSignal\ (Maybe\ \alpha)\ (DSignal\ \alpha)$$

A value *ExtDSignal Nothing* $\ddot{x}$ shall denote $\ddot{x}$, and a value *ExtDSignal* (*Just* x_0) $\ddot{x}$ shall denote $\ddot{x}$ with an additional event at t_0 that has value x_0.

3.1.2. Time

We have not yet specified what times and time differences are. The straightforward definition would be that *Time* is the type of real numbers, and *PTD* is the type of positive real numbers. However, this enables discrete signals that contain infinitely many events within a bounded time interval. An example is the discrete signal with the meaning $[(1 - i^{-1}, i) \mid i \leftarrow [2\,..]]$. We want to exclude such signals, since they result in typical problems related to supertasks [16].[1] Therefore, we define that *Time* is the type of all natural numbers, and *PTD* is the type of all positive integers. We further define that t_0, the start time of the program, is zero.

This definition might seem overly restrictive. It enforces that events can only occur at equally spaced times, and it fixes the start time of the program. However, these restrictions are not severe. Say we have an order-preserving bijection f from the set of real numbers onto itself. Now, we can interpret a real number t as the time $f\ t$ instead of the time t. This reinterpretation amounts to shifting, stretching and compressing the time line. By choosing an appropriate bijection f, we can map the natural numbers onto any unbounded, monotone sequence of real numbers. The elements of such a sequence can then act as actual event times, while the natural numbers used in signal meanings are just their representations. The signal combinator semantics presented in Section 3.2 is defined such that program behavior is independent of how event times are represented by natural numbers.

Stretching and compressing the time line has the consequence that a value Δt of type *PTD* does not uniquely represent a time difference anymore. The reason is that the result of $f\ (t + \Delta t) - f\ t$ may depend on t. However, this is not a problem for the FRP semantics. Every occurrence of a *PTD* value in the semantics indicates the length of a specific interval. For example in a discrete signal meaning $[(\Delta t_1, x_1), (\Delta t_2, x_2), \ldots]$, each Δt_i refers to the interval $(t_{i-1}, t_i]$, where $t_i = t_0 + (\Delta t_1 + \ldots + \Delta t_i)$ for any natural number i. We consider a *PTD* value that refers to an interval $(t, t']$ a representation of the time difference $f\ t' - f\ t$ only.

This interpretation of *PTD* values causes no problems because of the way *PTD* values are used in the signal combinator semantics. A value t of type *Time* and a value Δt of type *PTD* are only added if t represents the lower endpoint of the interval that Δt refers to. Likewise, two *PTD* values are only added if their intervals are adjacent. So if $[(\Delta t_1, x_1), (\Delta t_2, x_2), \ldots]$ is a discrete signal meaning, a sum $t_{i-1} + \Delta t_i$ or $\Delta t_i + \Delta t_{i+1}$ may occur in the semantics, while $t_i + \Delta t_i$ and $\Delta t_{i-1} + \Delta t_{i+1}$ may not.

So the moral of the story is that the concrete definition of *Time* and *PTD* is merely technical. The only visible effect of using natural numbers to represent event times is that only finitely many events can occur within a bounded time

[1] For example, the results of the *conjoin* combinator from Subsection 3.1.4 and the *switch* combinator from Subsection 3.2.5 may not be well-defined if such signals are allowed.

interval. If we take this restriction into account, we can keep the intuition that the values of type *Time* are elements of a continuous time line, and the values of type *PTD* are their positive differences.

3.1.3. Continuous Signals

The semantics of continuous signals is usually defined as follows:

type $[\![\mathit{CSignal}]\!]\ \alpha = \mathit{Time} \to \alpha$

The intention is that for each continuous signal $\tilde{x}$ and each time t, $[\![\tilde{x}]\!]\ t$ is the value of $\tilde{x}$ at time t.

Typically, a program does not know what happened before it started. Furthermore, certain time-varying values may not be defined or available at the program start, but only immediately afterwards. So it is not sensible to use *Time* as the domain of continuous signal meanings. We rather define $[\![\mathit{CSignal}]\!]$ as follows:

type $[\![\mathit{CSignal}]\!]\ \alpha = \mathit{PTD} \to \alpha$

For each continuous signal $\tilde{x}$ and each positive time difference Δt, $[\![\tilde{x}]\!]\ \Delta t$ is the value of $\tilde{x}$ at time $t_0 + \Delta t$. If we need a continuous signal that has a value also at t_0, we can use the type *ExtCSignal*:

data *ExtCSignal* α = *ExtCSignal* α (*CSignal* α)

A value *ExtCSignal* $x_0\ \tilde{x}$ denotes a continuous signal whose value at t_0 is x_0, and whose other values are given by $\tilde{x}$.

According to the last subsection, the *Time* type only covers a discrete selection of times. For each Δt, $t_0 + \Delta t$ is of type *Time*, so a continuous signal meaning is not defined on the complete continuous time scale. This does not matter though, because a program can access a continuous signal only at event occurences, and *Time* covers all times where an event may occur.

3.1.4. Segmented Signals

A segmented signal is similar to a continuous signal whose value only changes at discrete times. We call such times update times. Splitting a segmented signal at its update times yields a collection of constant fragments, which we call the segments of the signal.

A segmented signal is uniquely determined by its initial value and a sequence of events that occur at the update times and carry the respective new values. This is reflected in the definition of $[\![\mathit{SSignal}]\!]$:

type $[\![\mathit{SSignal}]\!]\ \alpha = (\alpha, [\![\mathit{DSignal}]\!]\ \alpha)$

A segmented signal with meaning $(x_0, [(\Delta t_1, x_1), (\Delta t_2, x_2), \ldots])$ starts with the value x_0 and changes its value at each time $t_0 + (\Delta t_1 + \ldots + \Delta t_i)$ to the respective value x_i.

Segmented signals can be converted into continuous signals. There is a signal combinator *conjoin* that achieves this conversion. Its meaning is defined as follows:

$$
\begin{aligned}
&[\![\mathit{conjoin}]\!] :: [\![\mathit{SSignal}]\!]\ \alpha \to [\![\mathit{CSignal}]\!]\ \alpha \\
&[\![\mathit{conjoin}]\!]\ (x_0, [\,]) &&= \mathit{const}\ x_0 \\
&[\![\mathit{conjoin}]\!]\ (x_0, (\Delta t, x) : \ddot{x}) &&= \lambda \Delta t' \to \mathbf{if}\ \Delta t' \leq \Delta t \\
&&&\quad \mathbf{then}\ x_0 \\
&&&\quad \mathbf{else}\ \ [\![\mathit{conjoin}]\!]\ (x, \ddot{x})\ (\Delta t' - \Delta t)
\end{aligned}
$$

The *conjoin* combinator is not only relevant for programming. It can also be used to overcome a weakness in the semantics of segmented signals. The meaning of a segmented signal does not tell us directly what value the signal has at some time. We can use the meaning of the corresponding continuous signal to obtain this information. The value of a segmented signal $\bar{x}$ at a time $t_0 + \Delta t$ is $[\![\mathit{conjoin}\ \bar{x}]\!]\ \Delta t$.

Note that in the second alternative of the $[\![\mathit{conjoin}]\!]$ definition, we check whether $\Delta t' \leq \Delta t$, not whether $\Delta t' < \Delta t$. This means that at an update time, a segmented signal still has the previous value. The new value comes into effect only immediately afterwards. To see that this makes sense, let us consider the start of the program. The time t_0 can be thought of as a kind of update time that has the value x_0 assigned to it. However, the signal has the value x_0 only immediately after t_0, since at t_0, it has no value at all.

The *conjoin* function is not injective, because the update times of a segmented signal cannot be recovered from its corresponding continuous signal. For example, a segmented signal $\bar{x}_1$ with a meaning $(x, [\,])$ is different from a segmented signal $\bar{x}_2$ with a meaning $(x, [(\Delta t, x)])$, but $[\![\mathit{conjoin}\ \bar{x}_1]\!]$ and $[\![\mathit{conjoin}\ \bar{x}_2]\!]$ are both $\mathit{const}\ x$. So segmented signals are not just specific continuous signals, but they contain more information than continuous signals. The *conjoin* function conjoins the segments of its argument, thus loosing the additional information about segment boundaries.

The reason for introducing segmented signals is access to that additional information in conjuncton with the guarantee that the value of a segmented signal is constant between update times. Consumers of segmented signals do not have to poll signal values constantly. It is enough for them to react at update times. In the network monitor application from Chapter 2, for example, the traffic volume display only needs to be updated when the user starts to monitor a different kind of traffic, or when a network packet enters or exits the system.

3.2. Signal Combinators

This section specifies the semantics of signal combinators. We first introduce the concept of signal suffixes in Subsection 3.2.1, since we use this concept for defining the meanings of signal combinators. Subsections 3.2.2 through 3.2.5 discuss the various combinators and their meanings.

3.2.1. Signal Suffixes

A suffix of a signal is the part of this signal that follows some time t with $t \geq t_0$. The time t is called the start time of the suffix. Suffix meanings have the same types as signal meanings, but they specify times relative to the start time of the suffix instead of t_0. For example, if $[(\Delta t_1, x_1), (\Delta t_2, x_2), \ldots]$ is the meaning of a discrete signal suffix with start time t, the i-th event of the suffix occurs at $t + (\Delta t_1 + \ldots + \Delta t_i)$, not $t_0 + (\Delta t_1 + \ldots + \Delta t_i)$.

For defining certain signal combinator meanings, we need an auxiliary function *pad*. This function transforms the meaning of a discrete signal suffix with some start time t into the meaning of a suffix that has an earlier start time t'. The suffix with start time t' contains the same events as the suffix with start time t. The only difference is that it also covers the interval $(t', t]$ with no events in it. The function *pad* is defined as follows:

$$
\begin{aligned}
&\mathit{pad} :: \mathit{PTD} \to [\![\mathit{DSignal}]\!]\ \alpha \to [\![\mathit{DSignal}]\!]\ \alpha \\
&\mathit{pad}\ \Delta t'\ [] \qquad\quad = [] \\
&\mathit{pad}\ \Delta t'\ ((\Delta t, x) : \ddot{x}) = (\Delta t' + \Delta t, x) : \ddot{x}
\end{aligned}
$$

The first argument of *pad* specifies the difference between the old and the new start time.

3.2.2. Discrete Signal Combinators

A discrete signal is similar to the list of its event values. This is reflected by the fact that a type $[\![\mathit{DSignal}]\!]\ \alpha$ is almost $[\alpha]$, the only difference being that in a value of type $[\![\mathit{DSignal}]\!]\ \alpha$, every value of type α is tagged with a time difference. The similarity between discrete signals and lists suggests a couple of discrete signal combinators that are analogs of list functions. Figure 3.1 defines the meanings of these combinators.

There is a problem with the definition of $[\![\mathit{filter}]\!]$. If a function f yields *False* for all event values of an infinite discrete signal meaning $\ddot{x}$, the expression $[\![\mathit{filter}]\!]\ f\ \ddot{x}$ yields $\bot$. However, we want it to yield []. More generally, an expression $[\![\mathit{filter}]\!]\ f\ \ddot{x}$ yields a result of the form $(\Delta t_1, x_1) : \ldots : (\Delta t_n, x_n) : \bot$ if $\ddot{x}$ ends in an infinite sequence of pairs $(\Delta t, x)$ with $\neg\ (f\ x)$, but we want $[\![\mathit{filter}]\!]\ f\ \ddot{x}$ to be $[(\Delta t_1, x_1), \ldots, (\Delta t_n, x_n)]$ in this case.

We can solve this problem with a different semantics for discrete signals. In this semantics, a meaning of a discrete signal is a function of type $\mathit{PTD} \to [(\mathit{PTD}, \alpha)]$. Applying such a meaning to a time difference Δt yields a list that only covers those events that occur within the finite time interval $(t_0, t_0 + \Delta t]$. The meaning of *filter* in this new semantics is the function $\lambda f\ \ddot{x} \to \lambda \Delta t \to [\![\mathit{filter}]\!]\ f\ (\ddot{x}\ \Delta t)$, where $[\![\mathit{filter}]\!]$ is the old meaning from Figure 3.1. Now, $[\![\mathit{filter}]\!]$ cannot produce unwanted $\bot$-values anymore, since it is only applied to finite lists.

While this alternative semantics removes unwanted bottoms, it has two drawbacks. The first one is that we cannot allow arbitrary functions of type $\mathit{PTD} \to [(\mathit{PTD}, \alpha)]$

$$
\begin{aligned}
&[\![map]\!] :: (\alpha \to \beta) \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \\
&[\![map]\!]\ f\ [] = [] \\
&[\![map]\!]\ f\ ((\Delta t, x) : \ddot{x}) = (\Delta t, f\ x) : [\![map]\!]\ f\ \ddot{x} \\
&[\![scanl1]\!] :: (\alpha \to \alpha \to \alpha) \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \alpha \\
&[\![scanl1]\!]\ \_\ [] = [] \\
&[\![scanl1]\!]\ f\ ((\Delta t, x) : \ddot{x}) = (\Delta t, x) : a\ x\ \ddot{x}\ \textbf{where} \\
&\quad a\ \_\ [] = [] \\
&\quad a\ y_0\ ((\Delta t, x) : \ddot{x}) = \textbf{let} \\
&\qquad\qquad y = f\ y_0\ x \\
&\qquad\quad \textbf{in}\ (\Delta t, y) : a\ y\ \ddot{x} \\
&[\![filter]\!] :: (\alpha \to Bool) \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \alpha \\
&[\![filter]\!]\ \_\ [] = [] \\
&[\![filter]\!]\ f\ ((\Delta t, x) : \ddot{x}) \mid f\ x = (\Delta t, x) : [\![filter]\!]\ f\ \ddot{x} \\
&\qquad\qquad\qquad\qquad \mid otherwise = pad\ \Delta t\ ([\![filter]\!]\ f\ \ddot{x}) \\
&[\![catMaybes]\!] :: [\![DSignal]\!]\ (Maybe\ \alpha) \to [\![DSignal]\!]\ \alpha \\
&[\![catMaybes]\!] = [\![map]\!]\ fromJust \circ [\![filter]\!]\ isJust \\
&[\![mapMaybe]\!] :: (\alpha \to Maybe\ \beta) \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \\
&[\![mapMaybe]\!]\ f = [\![catMaybes]\!] \circ [\![map]\!]\ f
\end{aligned}
$$

Figure 3.1.: Semantics of *DSignal* combinators that resemble functions on lists

as discrete signal meanings. A function $\ddot{x}$ of this type is only a valid meaning if for any positive time differences Δt and $\Delta t'$ with $\Delta t \leq \Delta t'$, $\ddot{x}\ \Delta t$ is the largest prefix $[(\Delta t_1, x_1), \ldots, (\Delta t_n, x_n)]$ of $\ddot{x}\ \Delta t'$ for which $\Delta t_1 + \ldots + \Delta t_n \leq \Delta t$. However, we cannot encode this additional restriction using Haskell types. The second drawback of the new semantics is that it makes the definitions of signal combinator meanings much more complex.

Because of these two drawbacks, we carry on with the original semantics. We add the exception that $[\![filter]\!]\ f\ \ddot{x}$ is $[]$ if $\neg\ (f\ x)$ for all elements $(\Delta t, x)$ of $\ddot{x}$. We just cannot formulate this as Haskell code, since Haskell cannot check whether all elements of an infinite list fulfill some condition.

We can merge two discrete signals $\ddot{x}_1$ and $\ddot{x}_2$ to get a discrete signal that generally contains all events from $\ddot{x}_1$ as well as $\ddot{x}_2$. If an event from $\ddot{x}_1$ occurs at the same time as an event from $\ddot{x}_2$, merging shall fuse these two events into a single one. The value of this single event shall be generated by applying a user-specified function to the values of the two events it is constructed from. We introduce a function *merge* that offers merging of discrete signals. It is even more powerful, since it allows the user to specify functions that are applied to event values which are not combined with other event values. The meaning of *merge* is given in Figure 3.2.

A discrete signal with finitely many events corresponds to a value of type

$$
\begin{array}{l}
[\![merge]\!] :: (\alpha \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\
\qquad\qquad (\beta \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\
\qquad\qquad (\alpha \to \beta \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\
\qquad\qquad ([\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \to [\![DSignal]\!]\ \gamma) \\
[\![merge]\!]\ \_\ r\ \ \_\ [] \qquad\qquad\qquad \ddot{x}_2 \qquad\qquad\qquad\qquad = [\![map]\!]\ r\ \ddot{x}_2 \\
[\![merge]\!]\ l\ \ \_\ \_\ \ddot{x}_1 \qquad\qquad\qquad [] \qquad\qquad\qquad\qquad = [\![map]\!]\ l\ \ddot{x}_1 \\
[\![merge]\!]\ l\ \ r\ \ b\ \ ((\Delta t_1, x_1) : \ddot{x}_1)\ ((\Delta t_2, x_2) : \ddot{x}_2) = \ddot{x}\ \textbf{where} \\
\qquad \ddot{x} = \textbf{case}\ compare\ \Delta t_1\ \Delta t_2\ \textbf{of} \\
\qquad\qquad\qquad LT\ \to (\Delta t_1, l\ x_1) \qquad : [\![merge]\!]\ l\ r\ b\ \ddot{x}_1 \qquad\qquad\qquad ((\Delta t_2', x_2) : \ddot{x}_2) \\
\qquad\qquad\qquad EQ\ \to (\Delta t_1, b\ x_1\ x_2) : [\![merge]\!]\ l\ r\ b\ \ddot{x}_1 \qquad\qquad\qquad \ddot{x}_2 \\
\qquad\qquad\qquad GT\ \to (\Delta t_2, r\ x_2) \qquad : [\![merge]\!]\ l\ r\ b\ ((\Delta t_1', x_1) : \ddot{x}_1)\ \ddot{x}_2 \\
\qquad \Delta t_1' :: PTD \\
\qquad \Delta t_1' = \Delta t_1 - \Delta t_2 \\
\qquad \Delta t_2' :: PTD \\
\qquad \Delta t_2' = \Delta t_2 - \Delta t_1
\end{array}
$$

Figure 3.2.: Semantics of signal merging

Map PTD α that contains a key–value pair $(\Delta t, x)$ for each event that occurs at time $t_0 + \Delta t$ and carries x as its value. There is no such correspondence for signals with infinitely many events, since the *Map* type only covers finite maps. So *Map PTD* is not suitable as a meaning of *DSignal*. Nevertheless, there are several discrete signal combinators that correspond to functions from Haskell's *Data.Map* module. Their meanings are shown in Figure 3.3. Apart from *empty*, all these combinators are based on merging.

3.2.3. Continuous Signal Combinators

CSignal is an applicative functor [20], which means that functions of arbitrary arity can be lifted to become functions over continuous signals. Lifting a function f of a type $\alpha_1 \to \ldots \to \alpha_n \to \beta$ results in a function of type

$$CSignal\ \alpha_1 \to \ldots \to CSignal\ \alpha_n \to CSignal\ \beta$$

whose meaning is

$$\lambda\tilde{x}_1\ \ldots\ \tilde{x}_n \to \lambda\Delta t \to [\![f]\!]\ (\tilde{x}_1\ \Delta t)\ \ldots\ (\tilde{x}_n\ \Delta t)\ .$$

Haskell's *Applicative* class contains the method *pure*, which is the lifting operator for "nullary functions", and the method $(\circledast)$, which is the result of lifting the function application operator (\$). For every arity n, the corresponding lifting operator $liftA_n$ can be derived from these two methods as follows:[2]

[2] Note that $\circledast$ is left associative.

$$
\begin{aligned}
&[\![empty]\!] :: [\![DSignal]\!]\ \alpha \\
&[\![empty]\!] = [] \\
&[\![unionWith]\!] :: (\alpha \to \alpha \to \alpha) \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \alpha \\
&[\![unionWith]\!]\ f = [\![merge]\!]\ id\ id\ f \\
&[\![union]\!] :: [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \alpha \\
&[\![union]\!] = [\![unionWith]\!]\ const \\
&[\![unionsWith]\!] :: (\alpha \to \alpha \to \alpha) \to [[\![DSignal]\!]\ \alpha] \to [\![DSignal]\!]\ \alpha \\
&[\![unionsWith]\!]\ f = foldl\ ([\![unionWith]\!]\ f)\ empty \\
&[\![unions]\!] :: [[\![DSignal]\!]\ \alpha] \to [\![DSignal]\!]\ \alpha \\
&[\![unions]\!] = foldl\ [\![union]\!]\ empty \\
&[\![intersectionWith]\!] :: (\alpha \to \beta \to \gamma) \to [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \to [\![DSignal]\!]\ \gamma \\
&[\![intersectionWith]\!]\ f\ \ddot{x}_1\ \ddot{x}_2 = [\![catMaybes]\!]\ \$\ \mathbf{let} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad l\ \_\ \ = Nothing \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r\ \_\ \ = Nothing \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad b\ x_1\ x_2 = Just\ (f\ x_1\ x_2) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\mathbf{in}\ [\![merge]\!]\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2 \\
&[\![intersection]\!] :: [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \to [\![DSignal]\!]\ \alpha \\
&[\![intersection]\!] = [\![intersectionWith]\!]\ const \\
&[\![differenceWith]\!] :: (\alpha \to \beta \to Maybe\ \alpha) \qquad\qquad\qquad\qquad \to \\
&\qquad\qquad ([\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \to [\![DSignal]\!]\ \alpha) \\
&[\![differenceWith]\!]\ f\ \ddot{x}_1\ \ddot{x}_2 = [\![catMaybes]\!]\ ([\![merge]\!]\ Just\ (const\ Nothing)\ f\ \ddot{x}_1\ \ddot{x}_2) \\
&[\![difference]\!] :: [\![DSignal]\!]\ \alpha \to [\![DSignal]\!]\ \beta \to [\![DSignal]\!]\ \alpha \\
&[\![difference]\!] = [\![differenceWith]\!]\ ((const \circ const)\ Nothing)
\end{aligned}
$$

Figure 3.3.: Semantics of *DSignal* combinators that resemble functions on maps

$$
\begin{aligned}
&[\![\mathit{fmap}]\!] :: (\alpha \to \beta) \to [\![\mathit{CSignal}]\!]\ \alpha \to [\![\mathit{CSignal}]\!]\ \beta \\
&[\![\mathit{fmap}]\!] = (\circ) \\
&[\![\mathit{pure}]\!] :: \alpha \to [\![\mathit{CSignal}]\!]\ \alpha \\
&[\![\mathit{pure}]\!] = \mathit{const} \\
&[\![(\circledast)]\!] :: [\![\mathit{CSignal}]\!]\ (\alpha \to \beta) \to [\![\mathit{CSignal}]\!]\ \alpha \to [\![\mathit{CSignal}]\!]\ \beta \\
&[\![(\circledast)]\!]\ \tilde{f}\ \tilde{x} = \lambda\Delta t \to (\tilde{f}\ \Delta t)\ (\tilde{x}\ \Delta t) \\
&[\![\mathit{sample}]\!] : [\![\mathit{DSignal}]\!]\ (\alpha \to \beta) \to [\![\mathit{CSignal}]\!]\ \alpha \to [\![\mathit{DSignal}]\!]\ \beta \\
&[\![\mathit{sample}]\!]\ []\qquad\quad \_ = [] \\
&[\![\mathit{sample}]\!]\ ((\Delta t, f) : \ddot{f})\ \tilde{x} = (\Delta t, f\ (\tilde{x}\ \Delta t)) : [\![\mathit{sample}]\!]\ \ddot{f}\ (\lambda\Delta t' \to \tilde{x}\ (\Delta t + \Delta t'))
\end{aligned}
$$

Figure 3.4.: Semantics of *CSignal* combinators

$$\mathit{liftA}_n\ f\ a_1\ \ldots\ a_n = \mathit{pure}\ f \circledast a_1 \circledast \ldots \circledast a_n$$

Since *Applicative* is a subclass of *Functor*, *fmap* must be defined for every applicative functor. However, applicative functor laws dictate that *fmap* is the lifting operator for unary functions. So it is not necessary to state the meaning of *fmap* for continuous signals, since it has to be $[\![(\circledast)]\!] \circ [\![\mathit{pure}]\!]$ anyway. Nevertheless, we say explicitely what $[\![\mathit{fmap}]\!]$ is. The meanings of *fmap*, *pure*, and $(\circledast)$ for continuous signals are defined in Figure 3.4.

Figure 3.4 also defines the meaning of a sampling combinator. This combinator fetches the value of a continuous signal whenever an event from some discrete signal occurs. The value of this event is combined with the value of the continuous signal and a new event that carries the result is generated.

The combinator *sample* always uses function application for combining an event value with the value of a continuous signal. However based on *sample*, we can define a sampling combinator that allows arbitrary combining functions:

$$
\begin{aligned}
&\mathit{sample}' :: (\alpha \to \beta \to \gamma) \to \mathit{DSignal}\ \alpha \to \mathit{CSignal}\ \beta \to \mathit{DSignal}\ \gamma \\
&\mathit{sample}'\ f\ \ddot{x}\ \tilde{y} = \mathit{map}\ f\ \ddot{x}\ `\mathit{sample}`\ \tilde{y}
\end{aligned}
$$

Note that the type of sample' is similar to the type of liftA_2, while the type of *sample* is similar to the type of $(\circledast)$. The definition of sample' is actually related to the technique for deriving the liftA_n functions from *pure* and $(\circledast)$. For liftA_1, we have the equation

$$\mathit{liftA}_1\ f\ a_1 = \mathit{pure}\ f \circledast a_1\ ,$$

while for liftA_2, we have

$$\mathit{liftA}_2\ f\ a_1\ a_2 = \mathit{pure}\ f \circledast a_1 \circledast a_2\ .$$

Since $\mathit{liftA}_1 = \mathit{fmap}$, it follows that

$$\mathit{liftA}_2\ f\ a_1\ a_2 = \mathit{fmap}\ f\ a_1 \circledast a_2\ ,$$

$$
\begin{aligned}
&[\![\mathit{fmap}]\!] :: (\alpha \to \beta) \to [\![\mathit{SSignal}]\!]\ \alpha \to [\![\mathit{SSignal}]\!]\ \beta \\
&[\![\mathit{fmap}]\!]\ f\ (x_0, \ddot{x}) = (f\ x_0, [\![\mathit{map}]\!]\ f\ \ddot{x}) \\
&[\![\mathit{pure}]\!] :: \alpha \to [\![\mathit{SSignal}]\!]\ \alpha \\
&[\![\mathit{pure}]\!]\ x_0 = (x_0, [\![\mathit{empty}]\!]) \\
&[\![(\circledast)]\!] :: [\![\mathit{SSignal}]\!]\ (\alpha \to \beta) \to [\![\mathit{SSignal}]\!]\ \alpha \to [\![\mathit{SSignal}]\!]\ \beta \\
&[\![(\circledast)]\!]\ (f_0, \ddot{f})\ (x_0, \ddot{x}) = \mathbf{let} \\
&\qquad\qquad \ddot{u}_f = [\![\mathit{map}]\!]\ (\lambda f \to \lambda(\_, x) \to (f, x))\ \ddot{f} \\
&\qquad\qquad \ddot{u}_x = [\![\mathit{map}]\!]\ (\lambda x \to \lambda(f, \_) \to (f, x))\ \ddot{x} \\
&\qquad\qquad \ddot{u}\ \ = [\![\mathit{unionWith}]\!]\ (\circ)\ \ddot{u}_f\ \ddot{u}_x \\
&\qquad\quad \mathbf{in}\ [\![\mathit{fmap}]\!]\ (\mathit{uncurry}\ (\$))\ \$\ [\![\mathit{scanl}]\!]\ (\mathit{flip}\ (\$))\ (f_0, x_0)\ \ddot{u} \\
&[\![\mathit{scanl}]\!] :: (\beta \to \alpha \to \beta) \to \beta \to [\![\mathit{DSignal}]\!]\ \alpha \to [\![\mathit{SSignal}]\!]\ \beta \\
&[\![\mathit{scanl}]\!]\ \_\ y_0\ [] \qquad\qquad = (y_0, []) \\
&[\![\mathit{scanl}]\!]\ f\ \ y_0\ ((\Delta t, x) : \ddot{x}) = \mathbf{let} \\
&\qquad\qquad\qquad\qquad (y, \ddot{y}) = [\![\mathit{scanl}]\!]\ f\ (f\ y_0\ x)\ \ddot{x} \\
&\qquad\qquad\qquad \mathbf{in}\ (y_0, (\Delta t, y) : \ddot{y})
\end{aligned}
$$

Figure 3.5.: Semantics of *SSignal* combinators

which corresponds to the fact that

$$\mathit{sample}'\ f\ \ddot{x}\ \tilde{y} = \mathit{map}\ f\ \ddot{x}\ `\mathit{sample}`\ \tilde{y}\ .$$

3.2.4. Segmented Signal Combinators

Like *CSignal*, *SSignal* is an applicative functor. The *conjoin* function from Subsection 3.1.4 is an applicative functor homomorphism from *SSignal* to *CSignal*, that is, for any function f of some arity n and any segmented signals $\bar{x}_1$ through $\bar{x}_n$, the equation

$$\mathit{conjoin}\ (\mathit{liftA}_n\ f\ \bar{x}_1\ \ldots\ \bar{x}_n) = \mathit{liftA}_n\ f\ (\mathit{conjoin}\ \bar{x}_1)\ \ldots\ (\mathit{conjoin}\ \bar{x}_n)$$

holds. A segmented signal $\mathit{liftA}_n\ f\ \bar{x}_1\ \ldots\ \bar{x}_n$ is updated whenever at least one of the $\bar{x}_i$ is updated. This specification of update times together with the homomorphism property uniquely defines how the applicative functor operators work for *SSignal*. The meanings of *fmap*, *pure*, and $(\circledast)$ are given formally in Figure 3.5.

In Subsection 3.2.2, we introduced the discrete signal combinator *scanl1*, which corresponds to the list function of the same name. There is also the list function *scanl*, which is more flexible than *scanl1*, but there is no sensible *scanl* analog for discrete signals. The reason is that *scanl* generates lists that are one element longer than the argument lists they are generated from, so that a *scanl* analog for discrete

signals would have to invent a new event time. However, there is a variant of *scanl* that turns discrete signals into segmented signals. This is because a segmented signal can be seen as a discrete signal with an additional value, the initial value of the segmented signal. Figure 3.5 also states the meaning of this *scanl* variant.

3.2.5. Switching

Switching makes it possible to compose a signal from sections of other signals. It is performed by the *switch* combinator, which has type $SSignal\ (S\ \alpha) \to S\ \alpha$ for every signal type constructor S. Let $\bar{s}$ be a signal of type $SSignal\ (S\ \alpha)$. If T is the time interval of some segment of $\bar{s}$, and s is the value of $\bar{s}$ during that interval, the signal $switch\ \bar{s}$ behaves like s during T.

For defining the meaning of *switch*, we use two helper functions *age* and *hop*. The *age* function shears off an initial part of a signal suffix meaning. If Δt is a positive time difference and $[\![s]\!]$ is the meaning of a signal suffix with a start time t, $age\ \Delta t\ [\![s]\!]$ is the meaning of the part of s that follows $t + \Delta t$. The *hop* function takes a signal suffix meaning $[\![s_0]\!]$, a positive time difference Δt, and another signal suffix meaning $[\![s]\!]$. If the start time of s_0 is t, then the start time of s has to be $t + \Delta t$. The *hop* function extracts the part of $[\![s_0]\!]$ that corresponds to the interval $(t, t + \Delta t]$ and appends $[\![s]\!]$ to it.

The implementation of *age* and *hop* varies between signal types. Therefore, we introduce a class *SignalMeaning* which has *age* and *hop* as its methods. It would be straightforward to design *SignalMeaning* such that its instances are the meanings $[\![S]\!]$ of signal type constructors. This is not possible, however, since these meanings are type aliases that are not fully applied. Therefore, we design the *SignalMeaning* class such that its instances are the types $[\![S]\!]\ \alpha$ where S is a signal type constructor and α is an arbitrary type. The class declaration is as follows:

```
class SignalMeaning m where
  age :: PTD → m → m
  hop :: m → PTD → m → m
```

The instance declarations for *SignalMeaning* are given in Figure 3.6. Based on *age* and *hop*, we can define the meaning of *switch* generically for all signal types. This is done in Figure 3.7.

3.3. Generators

In some FRP systems, signals are not first class. Instead, the programmer deals with so-called generators. At each time during program execution, a generator can generate a signal suffix that starts at that time. So essentially, a generator assigns a signal suffix with start time t to each time t with $t \geq t_0$. Generators are especially common in push-based FRP implementations, where they often arise naturally.

$$
\begin{aligned}
&\textbf{instance } \mathit{SignalMeaning}\ (\llbracket \mathit{DSignal} \rrbracket\ \alpha)\ \textbf{where}\\
&\quad \mathit{age}\ \_\ \ [] \qquad\qquad\qquad\qquad\qquad\qquad = []\\
&\quad \mathit{age}\ \Delta t'\ ((\Delta t, x) : \ddot{x}) \mid \Delta t > \Delta t' = (\Delta t - \Delta t', x) : \ddot{x}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \Delta t \equiv \Delta t' = \ddot{x}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \Delta t < \Delta t' = \mathit{age}\ (\Delta t' - \Delta t)\ \ddot{x}\\
&\quad \mathit{hop}\ []\qquad\qquad \Delta t'\ \ddot{y} \qquad\qquad\quad = \mathit{pad}\ \Delta t'\ \ddot{y}\\
&\quad \mathit{hop}\ ((\Delta t, x) : \ddot{x})\ \Delta t'\ \ddot{y} \mid \Delta t > \Delta t' = \mathit{pad}\ \Delta t'\ \ddot{y}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \Delta t \equiv \Delta t' = (\Delta t, x) : \ddot{y}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \Delta t < \Delta t' = (\Delta t, x) : \mathit{hop}\ \ddot{x}\ (\Delta t' - \Delta t)\ \ddot{y}\\
&\textbf{instance } \mathit{SignalMeaning}\ (\llbracket \mathit{CSignal} \rrbracket\ \alpha)\ \textbf{where}\\
&\quad \mathit{age}\ \Delta t'\ \tilde{x} = \lambda \Delta t \to \tilde{x}\ (\Delta t' + \Delta t)\\
&\quad \mathit{hop}\ \tilde{x}\ \Delta t'\ \tilde{y} = \lambda \Delta t \to \textbf{if } \Delta t \leq \Delta t' \textbf{ then } \tilde{x}\ \Delta t \textbf{ else } \tilde{y}\ (\Delta t - \Delta t')\\
&\textbf{instance } \mathit{SignalMeaning}\ (\llbracket \mathit{SSignal} \rrbracket\ \alpha)\ \textbf{where}\\
&\quad \mathit{age}\ \_\ \ (x_0, []) \qquad\qquad\qquad\qquad = (x_0, [])\\
&\quad \mathit{age}\ \Delta t'\ (x_0, (\Delta t, x) : \ddot{x}) \mid \Delta t > \Delta t' = (x_0, (\Delta t - \Delta t', x) : \ddot{x})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \Delta t \equiv \Delta t' = (x, \ddot{x})\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \mid \Delta t < \Delta t' = \mathit{age}\ (\Delta t' - \Delta t)\ (x, \ddot{x})\\
&\quad \mathit{hop}\ (x_0, \ddot{x})\ \Delta t'\ (y_0, \ddot{y}) = (x_0, h\ \ddot{x}\ \Delta t')\ \textbf{where}\\
&\qquad h\ []\qquad\qquad \Delta t' \qquad\qquad\quad = (\Delta t', y_0) : \ddot{y}\\
&\qquad h\ ((\Delta t, x) : \ddot{x})\ \Delta t' \mid \Delta t \geq \Delta t' = (\Delta t', y_0) : \ddot{y}\\
&\qquad\qquad\qquad\qquad\ \mid \Delta t < \Delta t' = (\Delta t, x) : h\ \ddot{x}\ (\Delta t' - \Delta t)
\end{aligned}
$$

Figure 3.6.: Instantiation of *SignalMeaning*

$$
\begin{aligned}
&\llbracket \mathit{switch} \rrbracket :: (\mathit{SignalMeaning}\ m) \Rightarrow \llbracket \mathit{SSignal} \rrbracket\ m \to m\\
&\llbracket \mathit{switch} \rrbracket\ (s_0, [])\ \ \ = s_0\\
&\llbracket \mathit{switch} \rrbracket\ (s_0, (\Delta t, s) : \ddot{s}) = \mathit{hop}\ s_0\ \Delta t\ (\llbracket \mathit{switch} \rrbracket\ \$\ \llbracket \mathit{fmap} \rrbracket\ (\mathit{age}\ \Delta t)\ (s, \ddot{s}))
\end{aligned}
$$

Figure 3.7.: Semantics of signal switching

$$
\begin{aligned}
&[\![merge^{\mathrm{g}}]\!] :: (\alpha \to \gamma) && \to \\
&\qquad\qquad (\beta \to \gamma) && \to \\
&\qquad\qquad (\alpha \to \beta \to \gamma) && \to \\
&\qquad\qquad ([\![Gen_{DSignal}]\!]\ \alpha \to [\![Gen_{DSignal}]\!]\ \beta \to [\![Gen_{DSignal}]\!]\ \gamma) \\
&[\![merge^{\mathrm{g}}]\!]\ l\ r\ b\ \ddot{x}^{\mathrm{g}}\ \ddot{y}^{\mathrm{g}} = \lambda t \to [\![merge]\!]\ l\ r\ b\ (\ddot{x}^{\mathrm{g}}\ t)\ (\ddot{y}^{\mathrm{g}}\ t)
\end{aligned}
$$

Figure 3.8.: Semantics of generator merging

For every signal type constructor S, we introduce a type constructor Gen_S such that a type $Gen_S\ \alpha$ covers all generators that generate suffixes of signals that have type $S\ \alpha$. We define $[\![Gen_S]\!]$ as follows:

$$\textbf{type}\ [\![Gen_S]\!]\ \alpha = Time \to [\![S]\!]\ \alpha$$

Remember that in Subsection 3.1.2, we defined t_0 and *Time* such that t_0 is the minimum of the *Time* type. So a generator does not assign signal suffixes to times that lie before the program start.

Like signals, generators can be produced and consumed. When a generator is consumed, it generates a signal suffix that starts at the time of consumption. This signal suffix is then used by the consumer to cause an impact on the real world. Since a generator may generate quite different signal suffixes depending on generation time, the effect caused by a consumer may heavily depend on the time of consumption.

For each signal combinator introduced in Section 3.2, there is an analogous combinator that works with generators instead of signals. Usually, the meaning of such a generator combinator f^{g} is gained by lifting the meaning of the underlying combinator f. This means that if f^{g} takes generators g_1 through g_n and other values x_1 through x_m as arguments and yields a generator g, applying $[\![g]\!]$ to a time t yields the same signal meaning as applying $[\![f]\!]$ to $[\![g_1]\!]\ t$ through $[\![g_n]\!]\ t$ and x_1 through x_m. Figure 3.8 demonstrates this principle by showing the meaning of a generator variant of *merge*.

The only signal combinator whose generator analog is not defined via lifting is *switch*. The semantics of generator switching is given in Figure 3.9. The fundamental difference between $[\![switch]\!]$ and $[\![switch^{\mathrm{g}}]\!]$ is that $[\![switch]\!]$ has to use the *age* function to adapt existing signal suffix meanings to later start times, while $[\![switch^{\mathrm{g}}]\!]$ directly generates suffix meanings that have the right start times.

As a consequence, the behavior of FRP applications may change fundamentally when turning from signals to generators. Take the network monitor application from Chapter 2, for example. Here, we visualize the value of the signal $\bar{v}$, which is constructed from $\bar{v}_{In}$, $\bar{v}_{Out}$, and $\bar{v}_{All}$ using *switch*. The signals $\bar{v}_{In}$, $\bar{v}_{Out}$, and $\bar{v}_{All}$ are computed from the producer outputs $\ddot{p}_{In}$ and $\ddot{p}_{Out}$, using the *volume* function to turn packet streams into traffic volume data.

Let us turn to a generator-based FRP system. Now everytime the user toggles

$$
\begin{aligned}
&[\![switch^{\mathrm{g}}]\!] :: (SignalMeaning\ m) \Rightarrow [\![Gen_{SSignal}]\!]\ (Time \to m) \to (Time \to m)\\
&[\![switch^{\mathrm{g}}]\!]\ \bar{g}^{\mathrm{g}} = \lambda t \to c\ t\ (\bar{g}^{\mathrm{g}}\ t)\ \textbf{where}\\
&\quad c\ t\ (g_0, [\,]) \qquad\quad = g_0\ t\\
&\quad c\ t\ (g_0, (\Delta t, g) : \ddot{g}) = hop\ (g_0\ t)\ \Delta t\ (c\ (t + \Delta t)\ (g, \ddot{g}))
\end{aligned}
$$

Figure 3.9.: Semantics of generator switching

the kind of traffic shown by the user interface, the program switches to a new traffic volume signal suffix. This suffix is computed from corresponding suffixes of $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ using the *volume* function internally. However, *volume* accumulates traffic volume starting with 0. As a consequence, the traffic volume display does not show the amount of traffic since the program start, but the amount of traffic since the last switch.

This example shows that using generators instead of signals might cause unexpected effects. In general, generators are a more complicated notion than signals. Therefore, we want to avoid generators, and provide signals as first class citizens instead. We will deal with this issue in Chapter 5.

4. Implementation

In this chapter, we present common techniques for implementing FRP. Such techniques can be divided into pull-based and push-based approaches. We structure our discussion according to this classification. Section 4.1 deals with pull-based implementations, and Section 4.2 discusses push-based ones. Our own contribution to FRP implementation techniques is described in Chapter 6.

4.1. Pull-Based Implementations

In a pull-based implementation, consumers poll data from signals. Signal representations have to accommodate this behavior, for example, by providing access to the current value of a continuous signal or by allowing a consumer to check whether a discrete signal has an event occurrence currently. A naïve pull-based implementation can be derived directly from the semantics. We just have to replace every occurrence of a signal type constructor meaning $[\![S]\!]$ by S and every occurrence of a signal combinator meaning $[\![f]\!]$ by f.

This approach has a small technical problem. As we already mentioned in Subsection 3.2.5, signal type constructor meanings are type aliases that are not fully applied. So they cannot be instances of classes like *Functor* and *Applicative*. This problem can be solved by defining signal type constructors as **newtype** wrappers instead:

$$\textbf{newtype}\ \mathit{DSignal}\ \alpha = \mathit{DSignal}\ [(\mathit{PTD}, \alpha)]$$
$$\textbf{newtype}\ \mathit{CSignal}\ \alpha = \mathit{CSignal}\ (\mathit{PTD} \to \alpha)$$
$$\textbf{newtype}\ \mathit{SSignal}\ \ \alpha = \mathit{SSignal}\ \ (\alpha, \mathit{DSignal}\ \alpha)$$

A downside of using **newtype** wrappers is that source code gets more verbose, since we often need to convert between **newtype** values and their internal representations. Therefore, we ignore the issue with class instantiation and continue defining *DSignal*, *CSignal*, and *SSignal* as type aliases.

In the following subsections, we successively improve the naïve implementation. We motivate each improvement by highlighting a specific deficiency that we want to remedy. Finally, we give concluding remarks on pull-based implementation approaches.

4.1.1. Absolute Time

In Subsection 3.1.1, we argued for using time differences in signal meanings instead of absolute times, since we can exclude illegal signal meanings that way. What is

$$
\begin{aligned}
&\mathit{sample} :: \mathit{DSignal}\ (\alpha \to \beta) \to \mathit{CSignal}\ \alpha \to \mathit{DSignal}\ \beta \\
&\mathit{sample}\ \ddot{f}\ \tilde{x} = \mathbf{let} \\
&\qquad\qquad r :: [\mathit{PTD}] \\
&\qquad\qquad r = \mathit{map}\ \mathit{fst}\ \ddot{f} \\
&\qquad\qquad a :: [\mathit{PTD}] \\
&\qquad\qquad a = \mathit{scanl1}\ (+)\ r \\
&\qquad\ \mathbf{in}\ \mathit{zip}\ r\ (\mathit{zipWith}\ (\lambda\Delta t\ f \to f\ (\tilde{x}\ \Delta t))\ a\ (\mathit{map}\ \mathit{snd}\ \ddot{f}))
\end{aligned}
$$

Figure 4.1.: Implementation of *sample* with sample time accumulation

an advantage for the semantics, is a burden for the implementation however. To see this, take a look at the *sample* combinator.

Applying *sample* to a discrete signal $[(\Delta t_1, f_1), (\Delta t_2, f_2), \ldots]$ and a continuous signal $\tilde{x}$ yields a discrete signal whose event values have the form $f_i\ (\tilde{x}\ (\Delta t_1 + \ldots + \Delta t_i))$, where i is the index of the respective event. Now, let us see how these values are generated. The definition of $[\![\mathit{sample}]\!]$ from Figure 3.4 replaces its argument $\tilde{x}$ by $\lambda\Delta t' \to \tilde{x}\ (\Delta t + \Delta t')$ in every recursion step. So the size of the continuous signal argument expressions grows linearly as sampling progresses, resulting in bad space behavior. Furthermore when the i-th value is sampled, the expression $\Delta t_1 + \ldots + \Delta t_i$ has to be computed, meaning that the time cost per sample grows linearly too.

The obvious solution is to reuse each sum $\Delta t_1 + \ldots + \Delta t_i$ in the computation of the next time difference sum $\Delta t_1 + \ldots + \Delta t_i + \Delta t_{i+1}$. This can be done as shown in Figure 4.1. The list a defined there contains the sums $\Delta t_1 + \ldots + \Delta t_i$. However, these sums are the differences between the sample times and t_0. So they are basically the absolute sample times, since they all refer to the same reference time (which even happens to be zero). So why not use absolute times in the first place if we calculate them later anyway?

We make the switch from relative to absolute time and define the signal type constructors as follows: [1]

$$
\begin{aligned}
&\mathbf{type}\ \mathit{DSignal}\ \alpha = [(\mathit{Time}, \alpha)] \\
&\mathbf{type}\ \mathit{CSignal}\ \alpha = \mathit{Time} \to \alpha \\
&\mathbf{type}\ \mathit{SSignal}\ \ \alpha = (\alpha, \mathit{DSignal}\ \alpha)
\end{aligned}
$$

Of course, the new definition of *DSignal* allows incorrect signal representations to be constructed, since it does not enforce that event times are strictly ascending and are greater than t_0. Therefore, the internal representation has to be hidden

[1] Here and in the following subsections, we continue to use type aliases for *DSignal*, *CSignal*, and *SSignal* whenever it is sensible.

from the user,[2] so that discrete signals can only be constructed by producers and signal combinators, which can be expected to only construct meaningful signal representations. Furthermore, we have to accept that representations of continuous signals might be partial functions, since they may be undefined at t_0.

4.1.2. Streams and Residuals

While the use of absolute time solves one space and time complexity problem, there is still another such problem. This other problem arises when continuous signals are composed from different segments, that is, if they are constructed by *conjoin* or *switch*. We discuss this issue by taking the example of *conjoin*, but the reasoning is similar for *switch*.

Let us look at an implementation of *conjoin*, which is directly derived from the definition of the meaning $[\![conjoin]\!]$, but uses absolute time instead of relative:

$$\begin{aligned} &conjoin :: SSignal\ \alpha \to CSignal\ \alpha \\ &conjoin\ (x_0, [])\ \ \ \ \ \ \ \ \ \ = const\ x_0 \\ &conjoin\ (x_0, (t, x) : \ddot{x}) = \lambda t' \to \textbf{if}\ t' \leq t\ \textbf{then}\ x_0\ \textbf{else}\ conjoin\ (x, \ddot{x})\ t' \end{aligned}$$

When a sample value of a signal *conjoin* $\bar{x}$ is calculated, the *conjoin* function iterates through the segments of $\bar{x}$ in order to find the segment that covers the sample time. So the time needed for sampling grows linearly with the number of segments that lie before the sample time. Furthermore, the list of all past update events has to be kept in memory, so that we can iterate through it again when the next value is sampled. Therefore, space usage grows linearly too.

It was already shortly after the invention of FRP that these problems became apparent. [8] There are two alternative implementations of continuous signals that solve these issues, the stream-based one and the residual-based one. [6] We will discuss them both in the remainder of this subsection.

In the stream-based implementation, a continuous signal is a function that turns a list of increasing sample times into the list of corresponding sample values:

$$\textbf{type}\ CSignal\ \alpha = [Time] \to [\alpha]$$

The idea is that during the execution of an FRP program, each continuous signal is applied only once, when the program starts. The argument of the signal is the list of all times at which the program needs to know the value of the signal.[3] So for each continuous signal, calculation of all sample values is done "in one go". As a result, information gathered during the calculation of earlier sample values can be used in the calculation of later sample values.

[2] This is not possible with type aliases, but it would be possible with **newtype** wrappers, of course.

[3] Of course, sample times might not be known at the start of the program. However, this is no problem, because lazy evaluation allows us to delay the generation of sample times until they are needed, which is when they have been reached.

$$
\begin{aligned}
&\mathit{conjoin} :: \mathit{SSignal}\ \alpha \to \mathit{CSignal}\ \alpha \\
&\mathit{conjoin}\ (x_0, [\,]) \qquad\quad = \mathit{map}\ (\mathit{const}\ x_0) \\
&\mathit{conjoin}\ (x_0, (t, x) : \ddot{x}) = \lambda ts' \to \mathbf{let} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (ts_0, ts) = \mathit{span}\ (\leq t)\ ts' \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{in}\ \mathit{map}\ (\mathit{const}\ x_0)\ ts_0 +\!\!+ \mathit{conjoin}\ (x, \ddot{x})\ ts
\end{aligned}
$$

Figure 4.2.: Stream-based implementation of *conjoin*

We use this feature in a stream-based implementation of *conjoin*, shown in Figure 4.2. Here, the gathered information is the remaining suffix of the segmented signal. Knowing this suffix, *conjoin* can check in constant time whether a new segment has been entered since the last sample time. Old information about the segmented signal is successively dropped and can be garbage-collected therefore. So the time cost per sample and the space cost are both constant.

Let us now look at the residual-based implementation of continuous signals. In this implementation, *CSignal* is defined as follows:[4]

$$\mathbf{type}\ \mathit{CSignal}\ \alpha = \mathit{Time} \to (\alpha, \mathit{CSignal}\ \alpha)$$

Applying a continuous signal to a sample time t yields the corresponding sample value and a so-called residual. The residual denotes the suffix of the signal that starts at t. So applying it to a time t' needs to yield meaningful results only if $t < t'$.

Say an FRP program needs to know the values of a continuous signal $\tilde{x}_0$ at times t_1, t_2, and so on, where $t_i < t_{i+1}$. For every t_i, the program calculates a sample value x_i and a residual $\tilde{x}_i$ by applying $\tilde{x}_{i-1}$ to t_i. As with the stream-based implementation, we can gather information during sampling and use it when calculating later sample values. The trick is to embed such information in the residuals. Figure 4.3 presents a residual-based implementation of *conjoin* that uses this technique. Again, the information that is gathered is the remaining suffix of the segmented signal. Time and space usage are the same as with the stream-based implementation.

4.1.3. Discrete Signals as Continuous Signals

Let us now look at a problem with the representation of discrete signals. Remember that we defined *DSignal* as follows:

$$\mathbf{type}\ \mathit{DSignal}\ \alpha = [(\mathit{Time}, \alpha)]$$

[4]The following definition of *CSignal* is not proper Haskell, because type synonym declarations cannot be recursive. We ignore this problem, since with **newtype** wrappers, it would vanish anyway.

$$
\begin{aligned}
&\mathit{conjoin} :: \mathit{SSignal}\ \alpha \to \mathit{CSignal}\ \alpha \\
&\mathit{conjoin}\ (x_0, [])\ = \mathbf{let} \\
&\qquad\qquad\qquad\qquad \tilde{x} = \mathit{const}\ (x_0, \tilde{x}) \\
&\qquad\qquad\qquad \mathbf{in}\ \tilde{x} \\
&\mathit{conjoin}\ (x_0, (t, x) : \ddot{x}) = \mathbf{let} \\
&\qquad\qquad\qquad\qquad \tilde{x} = \lambda t' \to \mathbf{case}\ \mathit{compare}\ t'\ t\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{LT}\ \to (x_0, \tilde{x}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{EQ}\ \to (x_0, \mathit{conjoin}\ (x, \ddot{x})) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{GT}\ \to \mathit{conjoin}\ (x, \ddot{x})\ t' \\
&\qquad\qquad\qquad\qquad \mathbf{in}\ \tilde{x}
\end{aligned}
$$

Figure 4.3.: Residual-based implementation of *conjoin*

Say we want to form the union of two non-empty discrete signals $(t_1, x_1) : \ddot{x}_1$ and $(t_2, x_2) : \ddot{x}_2$. In order to determine the first event of the union, we have to compare t_1 and t_2. If $t_1 \leq t_2$, the union will start with (t_1, x_1), if $t_1 > t_2$, it will start with (t_2, x_2). We have to further distinguish between the cases $t_1 < t_2$ and $t_1 \equiv t_2$, because in the latter case, the event (t_2, x_2) has to be excluded from the union. Note that the definition of $[\![\mathit{merge}]\!]$ in Figure 3.2 contains such a comparison of event times, the only difference being that $[\![\mathit{merge}]\!]$ uses relative time, while we use absolute time here.

If we want to compare two times, we usually have to know them both. However, we do not know the time of an event before this event occurs, that is, before the event time itself has been reached. So in order to know the first event of $\mathit{union}\ ((t_1, x_1) : \ddot{x}_1)\ ((t_2, x_2) : \ddot{x}_2)$, we may have to wait until the time $\mathit{max}\ t_1\ t_2$. However, the first event of the union already occurs at $\mathit{min}\ t_1\ t_2$, and we should know this event at this time, so that we are able to react upon it immediately.

A similar problem may arise if we form the union of a non-empty signal $(t, x) : \ddot{x}$ and the empty signal $[]$. The first event of the union can only be (t, x), but in order to know this, we have to determine that the second signal is in fact empty. Say the empty signal is produced by computing $\mathit{filter}\ f\ [(t_1, x_1), \ldots, (t_n, x_n)]$ where $\neg\ (f\ x_i)$ for all applicable i. Then, we have to wait until time t_n to know that the signal does not contain any events. If $t_n > t$, this means that we cannot deliver the first event of the union in time.

The problem is even worse if we filter an infinite signal where no events satisfy the filter predicate. In this case, we never know that the resulting signal is empty. A similar situation arises when a discrete signal mirrors a sequence of external events. It is usually not known whether a certain external event is the last one of its kind. For example, the discrete signal of all key presses can never be safely terminated, even if the user only makes a finite number of key presses, since we never know that the user is actually finished.

$$
\begin{aligned}
&\mathit{merge} :: (\alpha \to \gamma) \to \\
&\qquad (\beta \to \gamma) \to \\
&\qquad (\alpha \to \beta \to \gamma) \to \\
&\qquad (\mathit{DSignal}\ \alpha \to \mathit{DSignal}\ \beta \to \mathit{DSignal}\ \gamma) \\
&\mathit{merge}\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2 = \mathit{liftA}_2\ c\ \ddot{x}_1\ \ddot{x}_2\ \textbf{where} \\
&\quad c\ \mathit{Nothing}\ \ \mathit{Nothing}\ \ = \mathit{Nothing} \\
&\quad c\ \mathit{Nothing}\ \ (\mathit{Just}\ x_2) = \mathit{Just}\ (r\ x_2) \\
&\quad c\ (\mathit{Just}\ x_1)\ \mathit{Nothing}\ \ = \mathit{Just}\ (l\ x_1) \\
&\quad c\ (\mathit{Just}\ x_1)\ (\mathit{Just}\ x_2) = \mathit{Just}\ (b\ x_1\ x_2)
\end{aligned}
$$

Figure 4.4.: Implementation of *merge* based on *CSignal* lifting

A solution to all these problems lies in a completely different implementation of *DSignal*:

$$\textbf{type}\ \mathit{DSignal}\ \alpha = \mathit{CSignal}\ (\mathit{Maybe}\ \alpha)$$

A discrete signal is now represented by a continuous signal that assigns *Nothing* to a time t if there is no event at t, and *Just* x if there is an event with value x at t. The FRP implementation has to ensure that it fetches signal values at least at every event occurrence, because otherwise, events would be overlooked. [32]

Note that in a real-world setting, we would have to hide the implementation of *DSignal* again. Otherwise, a user could construct "discrete" signals with infinitely dense events. This could be done, for example, by applying *pure* to a value of the form *Just* x, thus creating a continuous signal that is constantly *Just* x.

Now, *merge* can be easily defined via the applicative functor operations of *CSignal*. This is shown in Figure 4.4. This new definition does not need to compare event times or to detect discrete signal termination. Let us illustrate this for the stream-based implementation of *CSignal*, although the reasoning is similar for the residual-based one.

In the stream-based approch, *CSignal* α is equivalent to $[\mathit{Time}] \to [\alpha]$, so *DSignal* α is $[\mathit{Time}] \to [\mathit{Maybe}\ \alpha]$ internally. The expression

$$\mathit{liftA}_2\ c\ \ddot{x}_1\ \ddot{x}_2\ \ ,$$

which is used in the implementation of *merge*, is equivalent to

$$\lambda ts \to \mathit{zipWith}\ c\ (\ddot{x}_1\ ts)\ (\ddot{x}_2\ ts)\ \ .$$

This means that the *Maybe* value of a signal *merge* $l\ r\ b\ \ddot{x}_1\ \ddot{x}_2$ at some time t is computed solely from the *Maybe* values $\ddot{x}_1$ and $\ddot{x}_2$ have at t. So we do not need information about the future to compute information about the present anymore.

The idea of implementing *DSignal* α as $[\mathit{Time}] \to [\mathit{Maybe}\ \alpha]$ was first published by Wan and Hudak. [32] A different implementation proposed by Elliott [6] represents discrete signals as lists of possible occurrences:

$$\textbf{type } \mathit{DSignal}\ \alpha = [(\mathit{Time}, \mathit{Maybe}\ \alpha)]$$

A possible occurrence $(t, \mathit{Nothing})$ signals the absence of an event at time t, while $(t, \mathit{Just}\ x)$ denotes an event with occurrence time t and value x.

In Elliotts implementation, merging two discrete signals $\ddot{x}_1$ and $\ddot{x}_2$ works without problems if for every possible occurrence in $\ddot{x}_1$ there is a possible occurrence in $\ddot{x}_2$ that has the same time, and vica versa. A conservative approach to fulfill this condition is to include possible occurrences for every sample time into each discrete signal. However, this has basically the same effect as implementing $\mathit{DSignal}\ \alpha$ as $[\mathit{Time}] \to [\mathit{Maybe}\ \alpha]$. Remember that the stream-based implementation applies continuous signals to the list of all sample times. So if discrete signals are implemented as continuous signals of *Maybe* values, *Maybe* values are computed for all sample times.

4.1.4. Signal Functions Instead of Signals

The stream-based and the residual-based FRP implementations allow for the recursive definition of signals. We demonstrate this with a recursive definition of a continuous signal whose meaning is the exponential function. For this aim, we assume the existence of two types *Real* and *PositiveReal*, covering all real numbers and all positive real numbers, respectively. Furthermore, we define that *Time* is *Real* and *PTD* is *PositiveReal*.

We first introduce a general-purpose integration function *integral*. We define the meaning of *integral* as follows:

$$[\![\mathit{integral}]\!] :: \mathit{Real} \to [\![\mathit{CSignal}]\!]\ \mathit{Real} \to [\![\mathit{CSignal}]\!]\ \mathit{Real}$$
$$[\![\mathit{integral}]\!]\ y_0\ \tilde{x} = \lambda\Delta t \to y_0 + \int_0^{\Delta t} (\tilde{x}\ \Delta t')\, d\Delta t'$$

Of course, we cannot implement this semantics precisely. We do not have the types *Real* and *PositiveReal* available, and even if we had, we could not compute exact integrals in general. So we switch from *Real* to *Double* and use an approximation method. A stream-based implementation of *integral* looks as follows:[5]

$$\begin{aligned}&\mathit{integral} :: \mathit{Double} \to \mathit{CSignal}\ \mathit{Double} \to \mathit{CSignal}\ \mathit{Double}\\ &\mathit{integral}\ y_0\ \tilde{x} = \lambda ts \to \mathit{scanl}\ (+)\ y_0\ \$\\ &\qquad\qquad \mathit{zipWith}\ (*)\ (\mathit{zipWith}\ (-)\ (\mathit{tail}\ ts)\ ts)\ (\tilde{x}\ ts)\end{aligned}$$

Using *integral*, we can implement the exponential function recursively:

$$\begin{aligned}&\mathit{exp} :: \mathit{CSignal}\ \mathit{Double}\\ &\mathit{exp} = \mathit{integral}\ 1\ \mathit{exp}\end{aligned}$$

While this definition is very elegant, it also suffers from bad time and space efficiency. We explain this for the stream-based implementation, although the problem also exists for a residual-based implementation.

[5] The *scanl* function used in the implementation of *integral* is the ordinary *scanl* function on lists, not the one on signals.

Using the implementation of *integral*, we can turn the above definition of *exp* into the following equation:

$$exp\ ts = scanl\ (+)\ 1\ \$\ zipWith\ (*)\ (zipWith\ (-)\ (tail\ ts)\ ts)\ (exp\ ts)$$

Let us see what happens if we compute the first n elements of *exp ts* for some fixed *ts*. In order to produce these n elements, the *scanl* function needs the first $n-1$ elements of its list argument. This argument has the form *zipWith* $(*)$ $(\ldots)$ $(exp\ ts)$. So to generate its first $n-1$ elements, we need the first $n-1$ elements of *exp ts*. Therefore, *exp* is applied to *ts* again and the first $n-1$ elements of the result are calculated. This requires another application of *exp* to *ts* for calculationg the first $n-2$ elements, and so on.

So we have a cascade of n applications *exp ts*. This means that space consumption increases linearly as sampling progresses. Generating a further element of *exp ts* causes each subordinate *exp ts* to also generate an element. So the time for producing a sample value also increases linearly. However, we want space consumption and time cost per sample to be constant.

The problem would vanish if the results of the different applications of *exp* to *ts* would be shared, but we do not know of any Haskell implementation that performs sharing for function applications. However, sharing is usually done for single variables. This allows us to transform the above equation into one that enables sharing of the generated list:

$$\begin{aligned} exp\ ts = \ & \textbf{let} \\ & \quad xs = scanl\ (+)\ 1\ \$\ zipWith\ (*)\ (zipWith\ (-)\ (tail\ ts)\ ts)\ xs \\ & \textbf{in}\ xs \end{aligned}$$

Now, it is the list *xs* of sample values that is defined recursively, not the *exp* signal itself. However, our aim was to define *exp* via the simple equation $exp = integral\ 1\ exp$. This equation dictates that *exp* is defined recursively, not the list of sample values. So the equation $exp = integral\ 1\ exp$ can never result in the *exp* implementation that employs sharing, even if we change the implementation of *integral*.

This trouble can be overcome by turning away from the idea that signals should be first class. If we base FRP on signal functions instead, we can come up with efficient recursion. We introduce a type constructor *CSignalFun* for continuous signal functions whose meaning is defined as follows:

$$\textbf{type}\ [\![CSignalFun]\!]\ \alpha\ \beta = [\![CSignal]\!]\ \alpha \to [\![CSignal]\!]\ \beta$$

The idea is to not implement continuous signal functions as ordinary functions, but in a way that is recursion-friendly. Now that we have *CSignalFun*, we remove the signal type constructors *DSignal*, *CSignal*, and *SSignal* from our FRP interface. Of course, this looks like a dramatic restriction of expressiveness. We will discuss this point in the next subsection.

There is a stream-based and a residual-based implementation of *CSignalFun*. The stream-based implementation is as follows: [22]

$$\textbf{type } \mathit{CSignalFun}\ \alpha\ \beta = [(\mathit{Time}, \alpha)] \to [\beta]$$

A function that represents a continuous signal function takes a list of pairs, each providing a sample time and the value that the input signal has at that time. From this list, the function computes the values that the output signal has at the given sample times. The residual-based implementation is as follows:

$$\textbf{type } \mathit{CSignalFun}\ \alpha\ \beta = \mathit{Time} \to \alpha \to (\beta, \mathit{CSignalFun}\ \alpha\ \beta)$$

Here, the representation of a continuous signal function takes a first sample time and the corresponding sample value of the input signal. It yields the respective sample value of the output signal together with a residual function. The residual function is responsible for turning signal suffixes into signal suffixes.

Note that with both implementations, the output value at some time can only depend on the values that have been sampled from the input signal until that time. As a consequence, signal functions cannot be represented exactly in general, but only approximated. Furthermore, only causal signal functions can be represented. This is no problem, since when processing signals online, we can only perform causal transformations anyway.

We can now define a dedicated fixpoint operator that works with *CSignalFun*. Its meaning is defined as follows:

$$\begin{aligned}&[\![\mathit{cSignalFix}]\!] :: [\![\mathit{CSignalFun}]\!]\ \alpha\ \alpha \to [\![\mathit{CSignalFun}]\!]\ ()\ \alpha\\ &[\![\mathit{cSignalFix}]\!]\ f = \mathit{const}\ (\mathit{fix}\ f)\end{aligned}$$

There is a stream-based and a residual-based implementation of this operator. We only discuss the stream-based one, although the residual-based implementation also achieves what we want.

$$\begin{aligned}&\mathit{cSignalFix} :: \mathit{CSignalFun}\ \alpha\ \alpha \to \mathit{CSignalFun}\ ()\ \alpha\\ &\mathit{cSignalFix}\ f = \lambda \mathit{is} \to \textbf{let}\\ &\qquad\qquad\qquad\qquad xs = f\ (\mathit{zip}\ (\mathit{map}\ \mathit{fst}\ \mathit{is})\ xs)\\ &\qquad\qquad\qquad \textbf{in}\ xs\end{aligned}$$

Note that the recursion is done in the definition of *xs*, so that sharing is enabled.

Now, we implement a variant of *integral* that is based on *CSignalFun*. We use the stream-based representation again:

$$\begin{aligned}&\mathit{integral} :: \mathit{Double} \to \mathit{CSignalFun}\ \mathit{Double}\ \mathit{Double}\\ &\mathit{integral}\ y_0 = \lambda \mathit{is} \to \textbf{let}\\ &\qquad\qquad\qquad\qquad (\mathit{ts}, \mathit{xs}) = \mathit{unzip}\ \mathit{is}\\ &\qquad\qquad\qquad \textbf{in}\ \mathit{scanl}\ (+)\ y_0\ \$\\ &\qquad\qquad\qquad\quad \mathit{zipWith}\ (*)\ (\mathit{zipWith}\ (-)\ (\mathit{tail}\ \mathit{ts})\ \mathit{ts})\ \mathit{xs}\end{aligned}$$

Finally, we implement *exp* using the fixpoint operator defined above:

```
exp :: CSignalFun () Double
exp = cSignalFix (integral 1)
```

Now, we take this implementation of *exp*, replace *cSignalFix* and *integral* by their definition, and simplify the result without changing its space and time characteristics. We arrive at the following equation:

```
exp is = let
           ts = map fst is
           xs = scanl (+) 1 $ zipWith (*) (zipWith (-) (tail ts) ts) xs
         in xs
```

This is almost the above definition of *exp* that uses sharing. The only difference is that the argument of *exp* is now of type $[(Time, ())]$ instead of $[Time]$, so that we have to drop the ()-values first.

The first detailed explaination of the issue with recursive definitions and its solution using signal functions was given by Liu and Hudak. [19] Our presentation heavily borrows from their work, although Liu and Hudak explained the problem and its solution using a residual-based implementation, while we use a stream-based implementation.

4.1.5. Optimization of the Signal Function Approach

As pointed out in the last subsection, moving from the original FRP interface to the interface based on *CSignalFun* seems to restrict expressiveness heavily. One point is that we cannot deal with discrete signals directly. We saw in Subsection 4.1.3 that we can represent discrete signals by continuous signals of *Maybe* values. So far, we used this approach only internally. However, we can also use it publicly, so that values of *CSignalFun* can also work with discrete signals. [22] A function from discrete signals to discrete signals, for example, has a type of the form *CSignalFun* (*Maybe* α) (*Maybe* β) now.

We noted in Subsection 4.1.3 that some continuous signals of *Maybe* values are not proper representations of discrete signals. As long as we represent discrete signals by continuous signals only internally, we can take care that no illegal data is constructed. Making the representation public opens the door for corrupted signal data. This is a severe weakness of this approach.

There is also no support for segmented signals in the *CSignalFun*-based FRP interface. The typical solution to this issue is to replace segmented signals by their corresponding continuous signals, that is, by the signals that would be created by applying *conjoin* to the respective segmented signals. However, this solution has a semantical problem too, since we loose information about update times, as explained in Subsection 3.1.4.

Another issue is that continuous signal functions only map single signals to single signals. However, *CSignalFun* can also represent functions whose arguments and

results are signal tuples. The reason is that for types α_1 through α_n,

$$(\llbracket \mathit{CSignal} \rrbracket\ \alpha_1, \ldots, \llbracket \mathit{CSignal} \rrbracket\ \alpha_n) \cong \llbracket \mathit{CSignal} \rrbracket\ (\alpha_1, \ldots, \alpha_n)\ .$$

So we can use values of a type *CSignalFun* $(\alpha_1, \ldots, \alpha_n)$ $(\beta_1, \ldots, \beta_m)$ to represent functions from $(\mathit{CSignal}\ \alpha_1, \ldots, \mathit{CSignal}\ \alpha_n)$ to $(\mathit{CSignal}\ \beta_1, \ldots, \mathit{CSignal}\ \beta_m)$.

Alas, the *CSignalFun* implementations presented in the last subsection are only well suited for functions that really take continuous signals to continuous signals. They behave badly in general when used for functions that involve discrete and segmented signals. For example, they do not make use of the fact that a segmented signal only changes at its update times. Instead, they force signal values to be recomputed even if no update time lies between the last and the current sample time.

It has turned out that such issues are hard or even impossible to resolve as long as we try to squeeze all FRP features into the corset of *CSignalFun*. Therefore, Sculthorpe and Nilsson [28, 29] have turned to an FRP approach that still uses functions as building blocks, but uses more fine-grained typing. Their approach is based on a type *SignalFun*, whose values denote functions. *SignalFun* has parameters that specify the domain and codomain of these functions. These parameters have to be of the form $(S_1\ \alpha_1, \ldots, S_n\ \alpha_n)$ where the S_i are either *DSignal* or *CSignal*. So one can distinguish between different types of signals and between single signals and tuples of signals.

Note that signals are still not first class. In fact, types of the forms *DSignal* α and *CSignal* α do not contain any values.[6] Signal type constructors are only used in the parameters of *SignalFun*. The implementation of *SignalFun* covers multiple alternative representations of signal functions. The signal type constructors are used to guide the choice of a concrete representation. That way, the representation of a signal function can accommodate for the types of input and output signals. The use of signal type constructors in *SignalFun* parameters also makes it possible to avoid the construction of corrupt discrete signals.

Besides domain and codomain, *SignalFun* can have further parameters, which state properties of signal functions. The types of signal function combinators can then specify constraints on such parameters. For example, a *SignalFun* parameter could specify whether the signal function is stateless, that is, whether its output at a certain time only depends on its input at the same time. The type of signal function composition would then encode the fact that the composition of two stateless functions is again stateless.

We can use additional parameters of *SignalFun* to identify functions that change their output and internal state only if the input changes. If the input of such a function stays constant, no computation is necessary for this function. We can use this fact to eliminate unnecessary recomputation of signal values.

[6] In one implementation, which uses the dependently-typed programming language Agda, *DSignal* and *CSignal* do not even construct types, but values.

4.1.6. Conclusions

In pull-based approaches, representations of signals or signal functions are declarative. In the simplest case, they directly mirror signal meanings. As a result, not only the FRP interface plays well with functional programming concepts, but also the implementation.

However, the declarative view on temporal phenomena often causes practical problems. For example, the definition of the meaning $[\![merge]\!]$ does not lead to an acceptable implementation of *merge*, as we saw in Subsection 4.1.3. Here, the background is that the semantics look at the whole time scale at once. So they do not take into account that we generally cannot know future event times and values. Furthermore, pull-based approaches do not support notifications about event occurrences and signal value updates. As a consequence, many pull-based implementations unnecessarily poll discrete signals for event occurrences and recompute signal values.

We saw that these problems can be solved without departing from the pull-based concept. In the next section we will present the push-based concept, which takes the abovementioned practical problems into account right from the start.

4.2. Push-Based Implementations

The essence of push-based FRP implementations is that consumers are notified whenever something interesting happens with the signals they consume. Consumers of discrete signals are notified about event occurrences, and consumers of segmented signals are notified about signal value updates. So these consumers do not have to poll their signals in order to detect events or value updates, respectively.

In Subsection 4.2.1, we introduce core constructs that support notification about event occurrences. Afterwards, we present different push-based implementation techniques and their problems. We give conclusions in Subsection 4.2.7.

4.2.1. Notification about Event Occurrences

In imperative languages, event notification is typically implemented via event handlers, sometimes also called listeners or callbacks. If a component wants to be notified about certain events, it registers a handler with the component that emits these events. The emitting component signals an event by calling all handlers registered with it. Values carried by events are transferred to the handlers as arguments. We use the concept of handler registration in the interfaces of push-based FRP systems to support event notification. The implementations from Subsections 4.2.2 and 4.2.3 use handler registration also internally.

Let us introduce the type *Handler* for describing event handlers in Haskell:

$$\textbf{type}\ \mathit{Handler}\ \alpha = \alpha \to \mathit{IO}\ ()$$

A handler takes a value carried by an event and turns it into an I/O action that describes the reaction to that event.

Handler registration could be described by a function that turns a handler into an I/O action that simply registers the handler. However, this gives us no means to unregister the handler at a later time. Therefore, we define the type of registration actions as follows:

$$\textbf{type}\ \mathit{Reg}\ \alpha = \mathit{Handler}\ \alpha \to \mathit{IO}\ (\mathit{IO}\ ())$$

Applying a registration action to a handler yields an I/O action of type $\mathit{IO}\ (\mathit{IO}\ ())$. This action registers the handler and returns an I/O action of type $\mathit{IO}\ ()$ that undoes the registration when called.

We use *Reg* in implementing producers and consumers. Let us look at producers first. As an example, take the producer *getInTraffic* from Chapter 2. We assume that an underlying low-level networking library provides us with a registration action *regInPacketHandler* of type *Reg Packet* through which we can be notified about incoming packets. We require that the FRP system contains a function *produce* for creating a discrete signal from a registration action:

$$\mathit{produce} :: \mathit{Reg}\ \alpha \to \mathit{IO}\ (\mathit{DSignal}\ \alpha)$$

Using *produce*, we can implement *getInTraffic* as follows:

$$\begin{aligned}&\mathit{getInTraffic} :: \mathit{IO}\ (\mathit{DSignal}\ \mathit{Packet})\\ &\mathit{getInTraffic} = \mathit{produce}\ \mathit{regInPacketHandler}\end{aligned}$$

The *produce* function can also be used for constructing parameterized producers. Say we want to distinguish between different network interfaces when listening for incoming packets. Let us assume the abovementioned low-level networking library exports a type *Interface* of network interfaces and a function *regInPacketHandlerForInterface* of type $\mathit{Interface} \to \mathit{Reg}\ \mathit{Packet}$. Applying *regInPacketHandlerForInterface* to an interface and a handler shall yield an I/O action that registers the handler such that it is only called when a packet enters the specified interface. We can define a parameterized producer as follows:

$$\begin{aligned}&\mathit{getInTrafficForInterface} :: \mathit{Interface} \to \mathit{IO}\ (\mathit{DSignal}\ \mathit{Packet})\\ &\mathit{getInTrafficForInterface} = \mathit{produce} \circ \mathit{regInPacketHandlerForInterface}\end{aligned}$$

A consumer of a discrete signal can be described by a handler that states what the consumer does in reaction to an event of the consumed signal. We want the FRP system to contain a function *consume* that registers a handler such that it is called at every event of a given discrete signal:

$$\mathit{consume} :: \mathit{DSignal}\ \alpha \to \mathit{Reg}\ \alpha$$

Say we want to create a consumer that takes a discrete signal of network packets. Whenever an event of this signal occurs, the consumer shall output the size of the respective packet on the terminal.[7] We implement this consumer as follows:

$$\begin{aligned}&\mathit{makePacketSizeWriter} :: \mathit{DSignal}\ \mathit{Packet} \to \mathit{IO}\ ()\\&\mathit{makePacketSizeWriter}\ \ddot{p} = \mathit{consume}\ \ddot{p}\ (\mathit{putStrLn} \circ \mathit{show} \circ \mathit{size}) \gg \mathit{return}\ ()\end{aligned}$$

We can also create parameterized consumers using the *consume* function. Say we want to output the packet sizes to a different stream. This can be implemented as follows:

$$\begin{aligned}&\mathit{makePacketSizeWriterForHandle} :: \mathit{Handle} \to \mathit{DSignal}\ \mathit{Packet} \to \mathit{IO}\ ()\\&\mathit{makePacketSizeWriterForHandle}\ h\ \ddot{p} = a\ \textbf{where}\\&\quad a = \mathit{consume}\ \ddot{p}\ (\mathit{hPutStrLn}\ h \circ \mathit{show} \circ \mathit{size}) \gg \mathit{return}\ ()\end{aligned}$$

In FRP systems that provide generators instead of signals, the functions *produce* and *consume* work with $\mathit{Gen}_{\mathit{DSignal}}$ values instead of *DSignal* values. Let us first look at *produce*:

$$\mathit{produce} :: \mathit{Reg}\ \alpha \to \mathit{IO}\ (\mathit{Gen}_{\mathit{DSignal}}\ \alpha)$$

An I/O action *produce r* yields a generator $\ddot{x}^{\mathrm{g}}$. Let $\ddot{x}$ be the discrete signal suffix that is generated by $\ddot{x}^{\mathrm{g}}$ at some time t. In order to listen for the events of $\ddot{x}$ via a handler h, we have to execute $r\ h$ at time t. So the registration time is used to determine the signal suffix to use.

Whenever a component emits an event, it calls all handlers that are currently registered with it using the event value as argument. No distinction is made regarding when the handlers were registered. So the different signal suffixes that are generated by $\ddot{x}^{\mathrm{g}}$ are mostly equal. Say $\ddot{x}_1$ and $\ddot{x}_2$ are two such suffixes where $\ddot{x}_1$ is generated earlier than $\ddot{x}_2$. Then $\ddot{x}_2$ is just a suffix of $\ddot{x}_1$, that is, during the lifetime of $\ddot{x}_2$, $\ddot{x}_1$ and $\ddot{x}_2$ behave identically.

The *consume* function turns a generator into a registration action:

$$\mathit{consume} :: \mathit{Gen}_{\mathit{DSignal}}\ \alpha \to \mathit{Reg}\ \alpha$$

Let $\ddot{x}^{\mathrm{g}}$ be a generator, $\ddot{x}$ be the discrete signal suffix generated by $\ddot{x}^{\mathrm{g}}$ at some time t, and h be a handler. Executing $\mathit{consume}\ \ddot{x}^{\mathrm{g}}\ h$ at time t registers the handler h such that it is called for every event of $\ddot{x}$. So the registration time determines the chosen signal suffix again.

4.2.2. Using Registration Actions in Generator Representations

Let us look at a simple way of implementing $\mathit{Gen}_{\mathit{DSignal}}$ and $\mathit{Gen}_{\mathit{SSignal}}$ in a push-based fashion. This approach was used in Grapefruit[8] in its early stages. Similar

[7] We do not take a consumer from the network monitor application to illustrate *consume*, since the only consumer in this application is *makeStringDisplay*, which consumes segmented signals, not discrete ones.

[8] See `http://grapefruit-project.org/`.

$$
\begin{aligned}
&\mathit{produce} :: \mathit{Reg}\ \alpha \to \mathit{IO}\ (\mathit{Gen}_{\mathit{DSignal}}\ \alpha)\\
&\mathit{produce} = \mathit{return}\\
&\mathit{consume} :: \mathit{Gen}_{\mathit{DSignal}}\ \alpha \to \mathit{Reg}\ \alpha\\
&\mathit{consume} = \mathit{id}
\end{aligned}
$$

Figure 4.5.: Implementation of *produce* and *consume* based on registration actions

techniques were also employed in other push-based FRP implementations like, for example, FranTk [25]. The presentation in this subsection is partially taken from an earlier work of us [13].

We represent a generator of discrete signal suffixes by the registration action that is created when *consume* is applied to the generator. So $\mathit{Gen}_{\mathit{DSignal}}$ is equivalent to *Reg*:

$$\textbf{type}\ \mathit{Gen}_{\mathit{DSignal}}\ \alpha = \mathit{Reg}\ \alpha$$

The implementation of *produce* and *consume* becomes trivial, as shown in Figure 4.5. For $\mathit{Gen}_{\mathit{SSignal}}$, we use an implementation that resembles the definition of $[\![\mathit{SSignal}]\!]$:

$$\textbf{type}\ \mathit{Gen}_{\mathit{SSignal}}\ \alpha = (\alpha, \mathit{Gen}_{\mathit{DSignal}}\ \alpha)$$

At each time t, a generator $(x, \ddot{x}^{\mathrm{g}})$ generates the segmented signal suffix whose initial value is x, and whose updates are given by the discrete signal suffix that is generated by $\ddot{x}^{\mathrm{g}}$ at time t.

Figure 4.6 shows the implementation of the generator combinators $\mathit{filter}^{\mathrm{g}}$ and $\mathit{scanl}^{\mathrm{g}}$, as well as the implementation of $\mathit{switch}^{\mathrm{g}}$ for the $\mathit{Gen}_{\mathit{DSignal}}$ case. Let us first look at the implementation of $\mathit{filter}^{\mathrm{g}}$. Applying a generator $\mathit{filter}^{\mathrm{g}}\ f\ \ddot{x}^{\mathrm{g}}$ to a handler h calls $\ddot{x}^{\mathrm{g}}$ to register a modified handler. This modified handler checks whether the predicate f is fulfilled for the event value x and calls the original handler h if it is.

A generator $\mathit{scanl}^{\mathrm{g}}\ f\ y_0\ \ddot{x}^{\mathrm{g}}$ provides update notifications via a generator $\ddot{y}^{\mathrm{g}}$. Say we want to listen for update events of a segmented signal suffix $\bar{y}$ generated by $\mathit{scanl}^{\mathrm{g}}\ f\ y_0\ \ddot{x}^{\mathrm{g}}$ at some time t. Then we have to apply $\ddot{y}^{\mathrm{g}}$ to a handler h at time t. The I/O action $\ddot{y}^{\mathrm{g}}\ h$ first creates a mutable variable for storing the current value of $\bar{y}$. Afterwards, it calls $\ddot{x}^{\mathrm{g}}$ to register a handler that updates the mutable variable and calls the original handler h with the new value.

Since the mutable variable is created and initialized during registration, accumulation of update values starts at the time of registration, which is the start time of $\bar{y}$. Furthermore, y_0 is the initial value of $\bar{y}$, since the result of $\mathit{scanl}^{\mathrm{g}}\ f\ y_0\ \ddot{x}^{\mathrm{g}}$ is $(y_0, \ddot{y}^{\mathrm{g}})$. So accumulation of signal values is done separately for each generated suffix, starting with y_0 at the start time of the respective suffix. This is in line with the semantics of generator combinators outlined in Section 3.3. As a consequence, a generator may generate signal suffixes that behave differently at the same time.

$$
\begin{aligned}
&\mathit{filter}^{\mathrm{g}} :: (\alpha \to \mathit{Bool}) \to \mathit{Gen}_{\mathit{DSignal}}\ \alpha \to \mathit{Gen}_{\mathit{DSignal}}\ \alpha \\
&\mathit{filter}^{\mathrm{g}}\ f\ \ddot{x}^{\mathrm{g}} = \lambda h \to \ddot{x}^{\mathrm{g}}\ (\lambda x \to \mathbf{if}\ f\ x\ \mathbf{then}\ h\ x\ \mathbf{else}\ \mathit{return}\ ()) \\
&\mathit{scanl}^{\mathrm{g}} :: (\beta \to \alpha \to \beta) \to \beta \to \mathit{Gen}_{\mathit{DSignal}}\ \alpha \to \mathit{Gen}_{\mathit{SSignal}}\ \beta \\
&\mathit{scanl}^{\mathrm{g}}\ f\ y_0\ \ddot{x}^{\mathrm{g}} = (y_0, \ddot{y}^{\mathrm{g}})\ \mathbf{where} \\
&\quad \ddot{y}^{\mathrm{g}} = \lambda h \to \mathbf{do} \\
&\qquad\qquad \vec{y} \leftarrow \mathit{newIORef}\ y_0 \\
&\qquad\qquad \ddot{x}^{\mathrm{g}}\ (\lambda x \to \mathbf{do} \\
&\qquad\qquad\qquad\qquad y \leftarrow \mathit{readIORef}\ \vec{y} \\
&\qquad\qquad\qquad\qquad \mathbf{let} \\
&\qquad\qquad\qquad\qquad\quad y' = f\ y\ x \\
&\qquad\qquad\qquad\qquad \mathit{writeIORef}\ \vec{y}\ y' \\
&\qquad\qquad\qquad\qquad h\ y') \\
&\mathit{switch}^{\mathrm{g}} :: \mathit{Gen}_{\mathit{SSignal}}\ (\mathit{Gen}_{\mathit{DSignal}}\ \alpha) \to \mathit{Gen}_{\mathit{DSignal}}\ \alpha \\
&\mathit{switch}^{\mathrm{g}}\ (\ddot{x}_0^{\mathrm{g}}, \ddot{\ddot{x}}^{\mathrm{g}}) = \lambda h \to \mathbf{do} \\
&\qquad\qquad\qquad\qquad u_0 \leftarrow \ddot{x}_0^{\mathrm{g}}\ h \\
&\qquad\qquad\qquad\qquad \vec{u}\ \ \leftarrow \mathit{newIORef}\ u_0 \\
&\qquad\qquad\qquad\qquad \hat{u}\ \ \leftarrow \ddot{\ddot{x}}^{\mathrm{g}}\ (\lambda \ddot{x}^{\mathrm{g}} \to \mathbf{do} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{join}\ (\mathit{readIORef}\ \vec{u}) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad u \leftarrow \ddot{x}^{\mathrm{g}}\ h \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{writeIORef}\ \vec{u}\ u) \\
&\qquad\qquad\qquad\qquad \mathit{return}\ (\mathit{join}\ (\mathit{readIORef}\ \vec{u}) \gg \hat{u})
\end{aligned}
$$

Figure 4.6.: Implementation of selected combinators based on registration actions

```
produce :: Reg α → IO (Gen_DSignal α)
produce = return ∘ const ∘ return

consume :: Gen_DSignal α → Reg α
consume ẍ^g = λh → do
                    t ← getCurrentTime
                    r ← ẍ^g t
                    r h
```

Figure 4.7.: Implementation of *produce* and *consume* with memoization support

Let us now look at *switch*$^{\mathrm{g}}$. When a generator *switch*$^{\mathrm{g}}$ $(\ddot{x}_0^{\mathrm{g}}, \dddot{x}^{\mathrm{g}})$ is applied to a handler h, it registers h using $\ddot{x}_0^{\mathrm{g}}$. Furthermore, it applies $\dddot{x}^{\mathrm{g}}$ to a handler that realizes a switch by unregistering the handler h and reregistering it using the generator that is switched to. The I/O action that undoes the last registration of h is stored in a mutable variable referenced by a pointer $\vec{u}$. Note that the I/O action *join* (*readIORef* $\vec{u}$) executes this stored I/O action.

4.2.3. Avoiding Unnecessary Recomputation

The implementation discussed in Subsection 4.2.2 has a performance problem. Computations like checking a predicate in *filter*$^{\mathrm{g}}$ and updating a signal value in *scanl*$^{\mathrm{g}}$ are done once per registered handler. If we register multiple handlers with the same generator at the same time, the different handlers monitor the same signal suffix. So the same computations are performed multiple times, which is an unnecessary repetition of work.

Sage [26, Section 7.4] presents a technique that allows us to avoid such superfluous recomputation. We implement $Gen_{DSignal}$ such that a generator of discrete signal suffixes is represented as a function from times to so-called setup actions:

$$\textbf{type } Gen_{DSignal}\ \alpha = Time \to IO\ (Reg\ \alpha)$$

Say $\ddot{x}^{\mathrm{g}}$ is a generator that generates a discrete signal suffix $\ddot{x}$ at some time t. If we want to listen for the events of $\ddot{x}$, we have to execute the setup action $\ddot{x}^{\mathrm{g}}\ t$ and use the resulting registration action to register a handler. This all has to happen at time t. The implementation of *produce* and *consume* is shown in Figure 4.7. The I/O action *getCurrentTime* used there has type *IO Time* and yields the current time.

A crucial point is that we have to execute two I/O actions in order to consume a generator, the setup action and the registration action that the setup action yields. We use this division into two I/O actions to separate computation from pure handler registration. We do everything related to computation in the setup action and let the registration action only perform the actual registration.

We then take care that computation-related code is executed only once per generator and start time. We realize this by turning the actual generator into a

```
scanl^g :: (β → α → β) → β → Gen_DSignal α → Gen_SSignal β
scanl^g f y0 ẍ^g = (y0, ÿ^g) where
  ÿ^g = memo (λt → do
                   r ← ẍ^g t
                   y⃗ ← newIORef y0
                   r (λx → do
                               y ← readIORef y⃗
                               writeIORef y⃗ (f y x))
                   return (λh → r (λ_ → do
                                          y′ ← readIORef y⃗
                                          h y′)))
```

Figure 4.8.: Implementation of $scanl^{\mathrm{g}}$ with memoization support

version that memoizes registration actions. We do so by applying a function *memo* of type

$$(Time \to IO\ \alpha) \to (Time \to IO\ \alpha)\ .$$

If applied to a function f of some type $Time \to IO\ \alpha$, *memo* creates a memo table that maps times to values of α and yields a function f'. An I/O action $f'\ t$ looks up t in the memo table. If there is a value for t, it returns it. Otherwise, it executes the I/O action $f\ t$, stores t together with the output of $f\ t$ in the memo table, and returns the output.

Figure 4.8 presents an implementation of $scanl^{\mathrm{g}}$. Note that the update generator $\ddot{y}^{\mathrm{g}}$ is constructed by applying *memo* to a function that turns a time t into a setup action. This setup action first executes the setup action for the argument generator and saves the resulting registration action r. Afterwards, it creates a mutable variable for storing the accumulated value. It uses r to register a handler that just updates this variable. Finally, it returns a registration action that takes a handler h and calls r to register a handler derived from h. The derived handler ignores the event value from the argument generator and just calls h with the accumulated value from the mutable variable. We assume that for each event, the FRP system calls the corresponding handlers in the order they were registered. This guarantees that the mutable variable has already been updated when the derived handler reads it.

4.2.4. A Problem with Simultaneous Events

A fundamental problem of the implementations shown in Subsections 4.2.2 and 4.2.3 is their inability to detect simultaneity of events. We demonstrate this problem for the simpler implementation of Subsection 4.2.2, where $Gen_{DSignal}$ is equivalent to *Reg*.

Let us try to implement $union^{\mathrm{g}}$. The following definition seems reasonable at a first glance:

$$union^{\mathrm{g}} :: Gen_{DSignal}\ \alpha \to Gen_{DSignal}\ \alpha \to Gen_{DSignal}\ \alpha$$
$$union^{\mathrm{g}}\ \ddot{x}_1^{\mathrm{g}}\ \ddot{x}_2^{\mathrm{g}} = \lambda h \to \ddot{x}_1^{\mathrm{g}}\ h \gg \ddot{x}_2^{\mathrm{g}}\ h$$

We just listen to the events of two discrete signal suffixes in order to listen to the events of their union. However, this approach does not handle simultaneous events correctly. Imagine we choose a single generator $\ddot{x}^{\mathrm{g}}$ and apply the generator $union^{\mathrm{g}}\ \ddot{x}^{\mathrm{g}}\ \ddot{x}^{\mathrm{g}}$ to a handler h at some time t. Then $\ddot{x}^{\mathrm{g}}$ is applied to h twice, so that h is called twice for every event of the suffix $\ddot{x}$ that is generated by $\ddot{x}^{\mathrm{g}}$ at t. This means that for each event of $\ddot{x}$, the suffix generated by $union^{\mathrm{g}}\ \ddot{x}^{\mathrm{g}}\ \ddot{x}^{\mathrm{g}}$ contains two events that conceptionally occur at the same time.

This problem cannot be solved just by changing the implementation of $union^{\mathrm{g}}$. Since the arguments of $union^{\mathrm{g}}$ are registration actions, the only thing we can do with them is to register handlers via them. Since these handlers run independently, fusing of simultaneous events is not possible. Analog problems exist with all combinators that are based on merging.

The inability to fuse simultaneous events also makes a correct implementation of the $(\circledast)$-combinator for $Gen_{SSignal}$ impossible. Say $\bar{y}$ is a segmented signal suffix generated by a generator $\bar{f}^{\mathrm{g}} \circledast \bar{x}^{\mathrm{g}}$ at some time t. Let $\bar{f}$ and $\bar{x}$ be the signal suffixes that are generated at t by $\bar{f}^{\mathrm{g}}$ and $\bar{x}^{\mathrm{g}}$, respectively. Say at some point after t, the value of $\bar{f}$ changes from f to f', and simultaneously, the value of $\bar{x}$ changes from x to x'. This means that the value of $\bar{y}$ has to change from $f\ y$ directly to $f'\ y'$. However, we cannot combine the update events of $\bar{f}$ and $\bar{x}$. So $\bar{f}$ will be updated before $\bar{x}$, or $\bar{x}$ will be updated before $\bar{f}$. As a result, $\bar{y}$ transiently adopts an unwanted intermediate value $f'\ x$ or $f\ x'$, respectively. This is called a glitch.

Cooper and Krishnamurthi developed a glitch-free implementation of segmented signals that does not rely on proper merging of discrete signals. Their technique is used in the Scheme library FrTime [4] and in the JavaScript-based FRP system Flapjax [21]. We do not discuss their implementation approach here. Instead, we focus on implementations that handle simultaneous events properly, which naturally leads to a glitch-free $(\circledast)$. We present two such approaches, one in the next subsection and one in Chapter 6.

4.2.5. An Implementation Based on Improving Times

In this subsection and the following one, we discuss Elliott's latest approach to implementing discrete and segmented signals [7], which is at the core of his Reactive library[9]. While this approach is a push-based one, it is derived from a pull-based implementation. Its key idea is an advanced definition of the *Time* type based on concurrency.

In Subsection 4.1.1, we implemented discrete signals as lists of events, where an event was represented by a pair of an occurrence time and an event value:

[9] See `http://haskell.org/haskellwiki/Reactive`.

$$\textbf{type } \mathit{DSignal}\ \alpha = [(\mathit{Time}, \alpha)]$$

In Subsection 4.1.3, we identified two problems with this implementation, both making it impossible to implement merging properly:

1. When all events of a finite signal have passed, we must be able to detect this immediately. However, this is often not possible. In the extreme case, we will never be able to tell that there are no more events.

2. We have to be able to compare two times t_1 and t_2 at time $\mathit{min}\ t_1\ t_2$. Usually, we can only compare them at time $\mathit{max}\ t_1\ t_2$.

We first change the implementation of *DSignal* such that only the second problem remains. Afterwards, we introduce the concept of improving times to solve the second problem.

Let us assume for a moment that discrete signals can only have infinitely many events. Then we can represent discrete signals by infinite lists of events:[10]

$$\begin{aligned}&\textbf{type } \mathit{InfList}\ \alpha\ \ = (\alpha, \mathit{InfList}\ \alpha)\\ &\textbf{type } \mathit{DSignal}\ \alpha = \mathit{InfList}\ (\mathit{Time}, \alpha)\end{aligned}$$

With this new definition, each discrete signal has the form $((t, x), \ddot{x})$. This is essentially a triple, which can also be represented by $(t, (x, \ddot{x}))$. However, the pair $(x, \ddot{x})$ is the representation of a segmented signal, whose initial value is x, and whose update signal is $\ddot{x}$. So we can define *DSignal* in terms of *SSignal*, while we can continue to define *SSignal* in terms of *DSignal*. This leads to a mutual recursive definition of *DSignal* and *SSignal*:

$$\begin{aligned}&\textbf{type } \mathit{DSignal}\ \alpha = (\mathit{Time}, \mathit{SSignal}\ \alpha)\\ &\textbf{type } \mathit{SSignal}\ \ \alpha = (\alpha, \mathit{DSignal}\ \alpha)\end{aligned}$$

Now, we extend the *Time* type with an additional value ∞ that represents a hypothetical time which lies infinitely far in the future. The pair $(\infty, \bar{x})$ represents a discrete signal whose first event "occurs at ∞", which essentially means that there will be no event for all time. So $(\infty, \bar{x})$ is a representation of the empty signal. Since the segmented signal $\bar{x}$ in a discrete signal $(t, \bar{x})$ is only of interested at time t and afterwards, there is no use for the $\bar{x}$ in $(\infty, \bar{x})$. So we can replace it by $\perp$ and thus represent the empty signal by $(\infty, \perp)$. We can use ∞ not just to represent the empty signal, but any finite signal. A signal with event values x_1 through x_n at times t_1 through t_n is encoded as $(t_1, (x_1, (\ldots, (t_n, (x_n, (\infty, \perp))) \ldots)))$.

Figure 4.9 shows an implementation of *merge* that is based on the new definition of *DSignal*. In contrast to the definition of $[\![\mathit{merge}]\!]$ in Figure 3.2, it does not contain separate equations for handling empty argument signals. The code that

[10] Note that in this subsection, we use recursive type synonyms, as we did in Subsection 4.1.2. We do so for simplicity again, ignoring the fact that type synonym declarations cannot be recursive actually.

$$
\begin{aligned}
&\mathit{merge} :: (\alpha \to \gamma) && \to \\
&\qquad (\beta \to \gamma) && \to \\
&\qquad (\alpha \to \beta \to \gamma) && \to \\
&\qquad (\mathit{DSignal}\ \alpha \to \mathit{DSignal}\ \beta \to \mathit{DSignal}\ \gamma) \\
&\mathit{merge}\ l\ r\ b\ (t_1, (x_1, \ddot{x}_1))\ (t_2, (x_2, \ddot{x}_2)) = \ddot{x}\ \mathbf{where} \\
&\quad \ddot{x} = \mathbf{case}\ \mathit{compare}\ t_1\ t_2\ \mathbf{of} \\
&\qquad\quad \mathit{LT} \to (t_1, (l\ x_1, \quad \mathit{merge}\ l\ r\ b\ \ddot{x}_1 \qquad\quad (t_2, (x_2, \ddot{x}_2)))) \\
&\qquad\quad \mathit{EQ} \to (t_1, (b\ x_1\ x_2, \mathit{merge}\ l\ r\ b\ \ddot{x}_1 \qquad\quad \ddot{x}_2)) \\
&\qquad\quad \mathit{GT} \to (t_2, (r\ x_2, \quad \mathit{merge}\ l\ r\ b\ (t_1, (x_1, \ddot{x}_1))\ \ddot{x}_2))
\end{aligned}
$$

Figure 4.9.: Implementation of *merge* without explicit handling of empty signals

normally handles non-empty signals also does the right thing if it encounters ∞ as an event time. So only the second of the abovementioned problems remains, albeit for a *Time* type that also contains ∞. We will solve this problem now.

Standard numeric types like *Double* and *Integer* denote domains that are flat. So if we use numbers for representing times, we can only distinguish between completely unknown times ($\bot$) and concrete times. What we need is a *Time* type that allows us to express partial knowledge about times. Specifically, we must be able to express that a time is greater than some reference time.

The solution is to apply the concept of improving values [1, 2] to times. The idea is that our knowledge about a time t may increase as time progresses. Before t has been reached, we know that t is larger than the current time. So we gain better and better lower bounds for t until we reach t. At t and afterwards, we know the concrete value of t. As a result, we can always successfully compare t with the current time. It follows that we can compare two times t_1 and t_2 at $\mathit{min}\ t_1\ t_2$. This is because $\mathit{min}\ t_1\ t_2$ is either t_1 or t_2. So either t_1 or t_2 is the current time at $\mathit{min}\ t_1\ t_2$, and the other t_i is the time that the current time is compared to.

As a special case, we are able to compare any time t with ∞ at t. The comparison result tells us that ∞ is greater than t, not that it is actually ∞. So when merging a non-empty signal $(t, \bar{x})$ with the empty signal, we do not determine that the second signal really has no events. We only determine that it has no events until t. This is enough to get the correct merge result, which means that the first of the abovementioned problems has vanished in fact.

Elliott developed an implementation of improving times based on concurrency. However, his approach is rather sophisticated. Therefore, we start with a simpler, albeit inefficient, implementation of *Time* that demonstrates the idea of improving times more clearly. We discuss Elliott's solution in the next subsection.

We define the type *Time* as follows:

$$\mathbf{data}\ \mathit{Time} = \mathit{Now} \mid \mathit{Later}\ \mathit{Time}$$

This definition resembles the well-known inductive definition of natural numbers,

$$
\begin{aligned}
&\mathit{compare} :: \mathit{Time} \to \mathit{Time} \to \mathit{Ordering} \\
&\mathit{compare}\ \mathit{Now}\ \ \mathit{Now}\ \ = \mathit{EQ} \\
&\mathit{compare}\ \mathit{Now}\ \ (\mathit{Later}\ \_) = \mathit{LT} \\
&\mathit{compare}\ (\mathit{Later}\ \_)\ \mathit{Now}\ \ = \mathit{GT} \\
&\mathit{compare}\ (\mathit{Later}\ t_1)\ (\mathit{Later}\ t_2) = \mathit{compare}\ t_1\ t_2
\end{aligned}
$$

Figure 4.10.: Simple improving times implementation of *compare*

where *Now* corresponds to zero, and *Later* corresponds to the successor function. However because of the non-strict semantics of Haskell, *Time* also covers a value for infinity:

$$
\begin{aligned}
&\infty :: \mathit{Time} \\
&\infty = \mathit{Later}\ \infty
\end{aligned}
$$

So a time is either a natural number or ∞.

Let Later^n denote $\mathit{Later} \circ \ldots \circ \mathit{Later}$ where *Later* occurs n times. A value of the form $\mathit{Later}^n\ t$ represents a time that is at least n. This is the key to expressing lower bounds of times. We produce each event time lazily. Whenever time advances by one, we generate a further application of *Later*. When the event finally occurs, we generate the terminating *Now*. If the expected event never occurs, we go on forever with producing *Later* applications. That way at each time n, we know either that the event time is $\mathit{Later}^{n+1}\ t$ for some t, or that it is $\mathit{Later}^m\ \mathit{Now}$ where $m \leq n$.

Figure 4.10 shows the implementation of *compare* for times. Let t_1 and t_2 be two times and t be their minimum. In order to compare t_1 and t_2, we only have to know the outer $t + 1$ data constructors of t_1 and t_2. However as explained above, we know these data constructors at time t. So time comparison can be done early enough.

We can demonstrate this by applying *compare* to arguments that contain $\bot$. Say we want to compare the *Time* values *Later Now* and *Later* (*Later Now*) at time 1. At this time, we do not know yet that the second value is *Later* (*Later Now*). We just know that it is *Later* (*Later t*) for some t. In order to show that a comparison at time 1 is possible, we replace t by $\bot$. The result of

$$\mathit{compare}\ (\mathit{Later}\ \mathit{Now})\ (\mathit{Later}\ (\mathit{Later}\ \bot))$$

is *LT*, not $\bot$. This proofs that when comparing *Later Now* with *Later* (*Later t*), the *compare* function does not touch t and therefore does not need to know anything about it.

Alas, the above definitions of *Time* and *compare* alone still give us no proper signal merging. Say we want to merge two discrete signals $(t_1, \bar{x}_1)$ and $(t_2, \bar{x}_2)$ to get a signal $(t, \bar{x})$. According to Figure 4.9, *merge* needs the result of *compare* $t_1\ t_2$ in order to tell us anything about $(t, \bar{x})$. So we know nothing about $(t, \bar{x})$ until time

min t_1 t_2, which is t. In particular, we know nothing about t itself until t. This is okay for reacting to the first event of $(t, \bar{x})$, which occurs at t. It means, however, that we do not get lower bounds for t before t. This can break comparisons of t with other times.

As an example, let us merge three discrete signals $(t_1, \bar{x}_1)$, $(t_2, \bar{x}_2)$, and $(t_3, \bar{x}_3)$ by first merging $(t_2, \bar{x}_2)$ and $(t_3, \bar{x}_3)$ into a signal $(t, \bar{x})$ and then merging $(t_1, \bar{x}_1)$ and $(t, \bar{x})$. We set the times t_i to the following values:

$$\begin{aligned} t_1 &= \mathit{Later}\ \mathit{Now} \\ t_2 &= \mathit{Later}\ (\mathit{Later}\ \mathit{Now}) \\ t_3 &= \mathit{Later}\ (\mathit{Later}\ (\mathit{Later}\ \mathit{Now})) \end{aligned}$$

For merging $(t_1, \bar{x}_1)$ and $(t, \bar{x})$, we have to be able to compare t_1 and t at time 1. However, we do not know anything about t before time 2, so that the needed comparison is not possible at 1.

We can illustrate this again using *Time* values that contain $\bot$. At time 1, we only know that t_2 and t_3 have the form *Later* (*Later* t'). So we set them to *Later* (*Later* $\bot$). This means that

$$\mathit{compare}\ t_2\ t_3 = \mathit{compare}\ (\mathit{Later}\ (\mathit{Later}\ \bot))\ (\mathit{Later}\ (\mathit{Later}\ \bot)) = \bot\ .$$

Since t depends on the result of *compare* t_2 t_3, t is $\bot$ too. So in order to merge $(t_1, \bar{x}_1)$ and $(t, \bar{x})$, we have to compare *Later Now* and $\bot$, which leads to $\bot$. So the merge result is completely undefined.

Figure 4.11 presents a slightly modified version of the *merge* implementation from Figure 4.9. With the new implementation, the event times in signals produced by *merge* do not depend on the outcomes of *compare* applications anymore. Instead, these times are calculated using the *min* function. We have to take care that *min* does not use *compare* internally. The default implementation of *min* is equivalent to

$$\lambda t_1\ t_2 \to \mathbf{if}\ \mathit{compare}\ t_1\ t_2 \equiv \mathit{GT}\ \mathbf{then}\ t_2\ \mathbf{else}\ t_1\ ,$$

so using it is not an option. Instead, we take a lazier implementation, which is shown in Figure 4.12. This implementation gives us proper lower bounds for all event times in *merge* results.

Let us demonstrate this with the above example of merging the three signals $(t_1, \bar{x}_1)$, $(t_2, \bar{x}_2)$, and $(t_3, \bar{x}_3)$. Remember that we set t_2 and t_3 to *Later* (*Later* $\bot$) to express the knowledge we have about these times at time 1. Merging $(t_2, \bar{x}_2)$ and $(t_3, \bar{x}_3)$ leads to the signal $(t, \bar{x})$, where

$$t = \mathit{min}\ t_2\ t_3 = \mathit{min}\ (\mathit{Later}\ (\mathit{Later}\ \bot))\ (\mathit{Later}\ (\mathit{Later}\ \bot)) = \mathit{Later}\ (\mathit{Later}\ \bot)\ .$$

Merging $(t_1, \bar{x}_1)$ and $(t, \bar{x})$ yields a signal whose first occurrence time is

$$\mathit{min}\ t_1\ t = \mathit{min}\ (\mathit{Later}\ \mathit{Now})\ (\mathit{Later}\ (\mathit{Later}\ \bot)) = \mathit{Later}\ \mathit{Now}\ .$$

$$
\begin{aligned}
&\mathit{merge} :: (\alpha \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \\
&\qquad\quad (\beta \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad\quad \to \\
&\qquad\quad (\alpha \to \beta \to \gamma) \qquad\qquad\qquad\qquad\qquad\quad \to \\
&\qquad\quad (\mathit{DSignal}\ \alpha \to \mathit{DSignal}\ \beta \to \mathit{DSignal}\ \gamma) \\
&\mathit{merge}\ l\ r\ b\ (t_1, (x_1, \ddot{x}_1))\ (t_2, (x_2, \ddot{x}_2)) = (\mathit{min}\ t_1\ t_2, \bar{x})\ \textbf{where} \\
&\quad \bar{x} = \textbf{case}\ \mathit{compare}\ t_1\ t_2\ \textbf{of} \\
&\qquad\quad \mathit{LT} \to (l\ x_1, \quad \mathit{merge}\ l\ r\ b\ \ddot{x}_1 \qquad\quad (t_2, (x_2, \ddot{x}_2))) \\
&\qquad\quad \mathit{EQ} \to (b\ x_1\ x_2, \mathit{merge}\ l\ r\ b\ \ddot{x}_1 \qquad\quad \ddot{x}_2) \\
&\qquad\quad \mathit{GT} \to (r\ x_2, \quad \mathit{merge}\ l\ r\ b\ (t_1, (x_1, \ddot{x}_1))\ \ddot{x}_2)
\end{aligned}
$$

Figure 4.11.: Implementation of *merge* based on improving times

$$
\begin{aligned}
&\mathit{min} :: \mathit{Time} \to \mathit{Time} \to \mathit{Time} \\
&\mathit{min}\ \mathit{Now}\ \ \_\ \ = \mathit{Now} \\
&\mathit{min}\ \_\ \ \mathit{Now}\ \ = \mathit{Now} \\
&\mathit{min}\ (\mathit{Later}\ t_1)\ (\mathit{Later}\ t_2) = \mathit{Later}\ (\mathit{min}\ t_1\ t_2)
\end{aligned}
$$

Figure 4.12.: Simple improving times implementation of *min*

So we know at time 1 that the first event of the final signal occurs at time 1.

The improving times implementation does not have typical problems of push-based implementations, because of its similarity to the pull-based implementation of Subsection 4.1.1. It supports signals instead of generators and allows for the combination of simultaneous events. Furthermore, event values are memoized, because they appear as part of a functional data structure. So we do not have the problem of unnecessary event value recomputation.

Alas, the definition of *DSignal* and *SSignal* shown in this subsection does not allow for an efficient implementation of switching. Whenever we switch to a discrete signal $\ddot{x}$ at some time t, we have to iterate through $\ddot{x}$ in order to find the first event that occurs after t. So we have to keep information about past events in memory only to ignore them when we switch. A related problem exists for segmented signals. These issues are similar to the problem we described in Subsection 4.1.2, and they result in analogous time and space efficiency problems. We present an efficient switching combinator in Chapter 6.

4.2.6. An Efficient Implementation of Improving Times

Besides the performance problems with switching that we mentioned at the end of the last subsection, there is also a performance problem with our implementation of improving times. Whenever time advances by one, each discrete signal producer has to extend the *Time* value of its next event by one application of *Later*. So performing a time step takes time linear in the number of discrete signal producers.

```
unamb :: α → α → α
x1 `unamb` x2 = unsafePerformIO (do
                                   x⃗ ← newEmptyMVar
                                   i1 ← forkIO (putMVar x⃗ $! x1)
                                   i2 ← forkIO (putMVar x⃗ $! x2)
                                   x ← takeMVar x⃗
                                   killThread i1
                                   killThread i2
                                   return x)
```

Figure 4.13.: Implementation of *unamb*

However, it should only take a constant amount of time. Elliott [7] introduced an implementation of improving times that does not have this problem. We describe his implementation in this subsection.[11]

Say *PrimTime* is an ordinary type that we can use for representing finite times, for example, *Integer* or *Double*. We derive *Time* from *PrimTime* as follows:

```
type Time = (PrimTime, PrimTime → Ordering)
```

A time is represented by a pair (p, c) where p is the *PrimTime* value that denotes the time, and c applied to a time p' compares the represented time with p'. So if we know the *PrimTime* representation of some time, we can form a *Time* representation using the following function:

```
fromPrimTime :: PrimTime → Time
fromPrimTime p = (p, compare p)
```

There are some further conditions a proper value of *Time* has to fulfill. Say (p, c) is a time representation. If evaluation of p is forced before the time p, the evaluating thread has to block until p and return the desired result only then. Otherwise, evaluation has to succeed immediately. On the other hand, evaluating $c\ p'$ for some p' has to block until p' or succeed immediately in case p' has already passed.

For implementing *compare* and *min*, we need a helper function *unamb*, which realizes "unambiguous choice". The implementation of *unamb* is shown in Figure 4.13. If the result of an expression x_1 `` `unamb` `` x_2 is requested, two threads are spawned for evaluating x_1 and x_2 concurrently. Each thread tries to put its result into an MVar, which is a one-bounded buffer. If one of the threads succeeds with this, its result is taken as the result of the *unamb* application, and both threads are killed.

[11] Actually, there are some subtle differences between Elliott's implementation and what we describe here. However, these differences are not crucial. We introduced them to make our presentation a bit simpler.

```
compare :: Time → Time → Ordering
compare (p1, c1) (p2, c2) = c1 p2 `unamb` m (c2 p1) where
  m :: Ordering → Ordering
  m LT = GT
  m EQ = EQ
  m GT = LT
```

Figure 4.14.: Efficient improving times implementation of *compare*

```
asAgree :: (Eq α) ⇒ α → α → α
x1 `asAgree` x2 = if x1 ≡ x2 then x1 else unsafePerformIO hang
```

Figure 4.15.: Implementation of *asAgree*

In order to retain referential transparency, the result of an *unamb* application must be uniquely defined. Therefore, one of the following conditions has to be fulfilled when calling *unamb*:

- Both *unamb* arguments yield the same result, possibly at different times.
- At least one of the arguments never yields a result, blocking evaluation forever.

We can compare two times (p_1, c_1) and (p_2, c_2) by applying c_1 to p_2. This gives us the comparison result at time p_2. However, we can also apply c_2 to p_1 and "mirror" the result to account for the different direction of comparison. This gives us the comparison result at time p_1. If we try both variants in parallel and take the result that is yielded earlier, comparison finishes at the earlier one of the two compared times. This is done in the implementation of *compare*, shown in Figure 4.14.

Implementing *min* needs a further utility function called *asAgree*. This function takes two arguments. If both arguments are equal, *asAgree* returns the single value that both arguments evaluate to. Otherwise, evaluation is blocked forever. The implementation of *asAgree* is shown in Figure 4.15. It uses an I/O action *hang* of type $IO\ \alpha$ that blocks the current thread eternally. There are multiple possibilities to implement *hang* such that it only consumes negligible resources. Elliott gives one based on *threadDelay*.

Figure 4.16 shows the implementation of *min*. We use *compare* to check whether the first time is smaller than the second time. The variables p and c are set to either p_1 and c_1 or p_2 and c_2, depending on the outcome of this comparison. We do not get a comparison result before the time $min\ p_1\ p_2$, which is p. This is okay for computing p itself. However, an expression $c\ p'$ has to be able to yield its result at p'. If p' is smaller than p, we have a problem.

Therefore, we derive a comparison function $\hat{c}$ from c. Running $\hat{c}$ with an argument p' computes $c\ p'$, but evaluates also $c_1\ p'$ and $c_2\ p'$. If both $c_1\ p'$ and $c_2\ p'$

$$
\begin{aligned}
&\mathit{min} :: \mathit{Time} \to \mathit{Time} \to \mathit{Time}\\
&\mathit{min}\ (p_1, c_1)\ (p_2, c_2) = (p, \hat{c})\ \textbf{where}\\
&\quad l :: \mathit{Bool}\\
&\quad l = \mathit{compare}\ (p_1, c_1)\ (p_2, c_2) \equiv \mathit{LT}\\
&\quad p :: \mathit{PrimTime}\\
&\quad p = \textbf{if}\ l\ \textbf{then}\ p_1\ \textbf{else}\ p_2\\
&\quad c :: \mathit{PrimTime} \to \mathit{Ordering}\\
&\quad c = \textbf{if}\ l\ \textbf{then}\ c_1\ \textbf{else}\ c_2\\
&\quad \hat{c} :: \mathit{PrimTime} \to \mathit{Ordering}\\
&\quad \hat{c}\ p' = c\ p'\ \text{`}\mathit{unamb}\text{`}\ (c_1\ p'\ \text{`}\mathit{asAgree}\text{`}\ c_2\ p')
\end{aligned}
$$

Figure 4.16.: Efficient improving times implementation of *min*

$$
\begin{aligned}
&\mathit{consume} :: \mathit{DSignal}\ \alpha \to \mathit{Reg}\ \alpha\\
&\mathit{consume}\ \ddot{x} = \lambda h \to \textbf{let}\\
&\qquad\qquad a\ ((p, \_), (x, \ddot{x})) = (p\ \text{`}\mathit{seq}\text{`}\ h\ x) \gg a\ \ddot{x}\\
&\qquad\quad \textbf{in}\ \mathit{forkIO}\ (a\ \ddot{x}) \gg\!= \mathit{return} \circ \mathit{killThread}
\end{aligned}
$$

Figure 4.17.: Efficient improving times implementation of *consume*

finish before $c\ p'$ and yield the same result, their common result is taken. This is okay, since

$$c\ p' \equiv c_1\ p' \vee c\ p' \equiv c_2\ p'$$

according to the definition of c and thus

$$c\ p' \equiv c_1\ p' \equiv c_2\ p' \ .$$

Both $c_1\ p'$ and $c_2\ p'$ can yield a result at p', so $\hat{c}$ is able to yield its result at p' too.

Figure 4.17 presents an implementation of *consume*. Registering a handler h with a discrete signal $\ddot{x}$ spawns a new thread, which walks through $\ddot{x}$. For every event, it forces the evaluation of the *PrimTime* value that represents the event time. Thus, the thread waits until this time has been reached. Afterwards, the event is handled. The unregistration action that *consume* returns just kills the thread.

Alas, implementing *produce* is not so easy as implementing *consume*. In his paper [7], Elliott does not elaborate on how to produce discrete signals. A look into the source code of Reactive[12] reveals that discrete signal production is very involved. Furthermore, the handling of improving times in Reactive is more complicated than what is shown here and in Elliott's paper. This is necessary to avoid certain issues

[12]This source code is available via `http://hackage.haskell.org/package/reactive`.

like subtle performance problems and problems when interacting with Haskell's exception mechanism.

In addition, the use of concurrency for implementing improving times causes a semantical problem. Since the threads of different consumers run asynchronously, it is not guaranteed that events are handled in the order they occur. Say there are two consumers that both wait for a next event to handle. Say the event for the first consumer occurs earlier. However, the thread of the first consumer has to be unblocked before this event can be handled. In the meantime, the event for the second consumer might occur, and it could be that the second consumer's thread gets unblocked and the event handled before the first consumer's thread gets unblocked. We do not know of any approach to solve this problem.

4.2.7. Conclusions

In push-based FRP systems, events and signal updates trigger reactions directly, which improves scalability. However, a naïve push-based implementation, as shown in Subsection 4.2.2, has several drawbacks:

- The FRP system does not provide signals, but only generators.
- Computations are repeated unnecessarily if the same signal suffix is used by more than one consumer.
- Simultaneous events cannot be combined during merging.

We have discussed two advanced push-based approaches. While the solution by Sage only solves the problem of repeated computation, the solution by Elliott solves all three issues. However, Elliott's FRP implementation adds problems of its own, namely, high code complexity, the possibility of out-of-order event handling, as well as time and space inefficiencies in the context of switching. In the following two chapters, we show how these problems can be overcome without reintroducing the drawbacks listed above.

5. Start Time Consistency

As noted at the end of Subsection 4.2.5, the improving times implementation of FRP does not allow for an efficient switching combinator. The reason is that a signal starts at t_0, but when we switch to a signal at some time t, we are only interested in the suffix of the signal that starts at t. So data about the signal's behavior during the interval $(t_0, t]$ is unnecessarily kept in memory and has to be skipped when switching.

FRP systems that are based on generators do not have this problem. When they perform a switch, a generator generates the suffix that has to be switched to. This suffix starts at the switching time. So it contains no superfluous data about the past that has to be skipped. This advantage comes at a price, though. As we pointed out in Section 3.3, the additional semantic complexity of generators makes the life of the user more difficult.

In this chapter, we present a solution to this dilemma. In Section 5.1, we show how we can modify legacy FRP systems such that they provide signal suffixes as their first class building blocks.[1] Our techniques rely on a constraint on the use of signal suffixes called start time consistency. Section 5.2 shows how we can statically enforce start time consistency using the type system. We finally explain in Section 5.3 how switching can be integrated into an FRP system with static start time consistency checks. Note that Sections 5.2 and 5.3 are largely taken from an earlier work of the author [13].

5.1. Signal Suffixes and Start Time Consistency

Our goal is to provide signal suffixes as first class citizens. However, we want to enforce that signal suffixes are only used in a start-time-consistent manner. This means that observation of a signal suffix only starts at the start time of this suffix. In particular, the following conditions have to be met:

- Switching to a signal suffix is only done at the start time of this suffix. If we want to switch to a signal suffix after its start time, we have to explicitely trim it such that it becomes a suffix that starts at the switching time.[2]

- Only signal suffixes with start time t_0, that is, ordinary signals are consumed.

[1] Note that this means that signals are first class too, since signals are just signal suffixes that start at t_0.

[2] This trimming corresponds to what the *age* function from Subsection 3.2.5 does.

The reason for the second restriction is that consumption is assumed to happen at the start of the program.

Let us now look at how we can transform existing FRP systems into systems that are based on signal suffixes. Say we start with an FRP system that provides signals instead of generators. We use the representations of signals to represent signal suffixes. We only have to respect the following points:

- Suffix representations do not contain data that refers to the time span until the start time of the suffix.
- If signal representations include times relative to t_0, such times have to be relative to the start times of the respective signal suffixes now.

The first point ensures that switching is efficient, since obsolete data does not have to be stored and skipped.

We can also transform an FRP system that is based on generators into a suffix-based system. We simply represent each suffix s by a generator g. Start time consistency ensures that g generates suffixes only at a single time, the time that is declared start time of s. So while a generator denotes a family of suffixes, g can only generate a single suffix, which is considered the suffix s that g represents.

5.2. Enforcing Start Time Consistency

We want to enforce start time consistency statically. Therefore, we use the type system to specify properties about start times. We then use the type checker to check for start time consistency at compile time. Our solution is inspired by the technique that makes Haskell's *ST* monad safe. We first review this technique and then adapt it to our needs.

5.2.1. Secure Handling of Stateful Computations

The *ST* monad [17] makes it possible to implement computations that appear to be purely functional from the outside, but use mutable state internally. A value of a type $ST\ \sigma\ \alpha$ denotes a stateful computation that yields a value of type α. Such a computation usually cannot interact with the outside world, but it can create, read from, and write to mutable variables. A value of a type $STRef\ \sigma\ \alpha$ is a reference to a mutable variable that can hold values of type α.

If we want a stateful computation to appear pure to the outside, the computation must not share mutable variables with other computations. If two computations share variables, one computation could change the behavior of the other one by modifying these variables. The σ-parameters of *ST* and *STRef* are used to prevent variable sharing.

The σ-parameter of *ST* is a phantom type parameter, which means that values of $ST\ \sigma\ \alpha$ do not deal with values of σ. Instead, σ denotes the set of all mutable variables that the stateful computation creates or uses. The σ-parameter of *STRef*

$$
\begin{array}{ll}
\mathit{newSTRef} & :: \alpha \to \qquad\qquad\quad \mathit{ST}\ \sigma\ (\mathit{STRef}\ \sigma\ \alpha) \\
\mathit{readSTRef} & :: \mathit{STRef}\ \sigma\ \alpha \to \quad \mathit{ST}\ \sigma\ \alpha \\
\mathit{writeSTRef} & :: \mathit{STRef}\ \sigma\ \alpha \to \alpha \to \mathit{ST}\ \sigma\ ()
\end{array}
$$

Figure 5.1.: Basic *STRef* operators

is also a phantom parameter. It denotes the set to which the referenced variable belongs. If a mutable variable is created or used by a stateful computation, the reference to this variable and the computation have to use the same σ-parameter. The types of the basic *STRef* operators enforce this by using the same type variable as parameters of *ST* and *STRef*. The type signatures of these operators are shown in Figure 5.1.

There is a function *runST* that turns a stateful computation into a seemingly pure computation. If c is a stateful computation, evaluating *runST* c runs c and yields its result. The type of *runST* is $(\forall\sigma.\mathit{ST}\ \sigma\ \alpha) \to \alpha$. The fact that it contains a universally quantified type to the left of a function arrow means that it uses higher-rank polymorphism.

The universal quantification enforces that *runST* can only be applied to stateful computations whose σ-parameters are arbitrary. These are the computations that do not share mutable variables with other computations. To see this, take two computations with σ-parameters σ_1 and σ_2, respectively. If both share a variable whose reference has a σ-parameter σ_r, the equations $\sigma_1 = \sigma_r$ and $\sigma_2 = \sigma_r$ hold. It follows that $\sigma_1 = \sigma_2$. So σ_1 and σ_2 depend on each other and cannot be chosen arbitrarily therefore.

ST and *STRef* can also be used to implement *IO* and its corresponding reference type *IORef* by setting σ-parameters to a fixed type that denotes the "real world":

data *RealWorld*
type $\mathit{IO}\ \alpha \quad = \mathit{ST}\ \mathit{RealWorld}\ \alpha$
type $\mathit{IORef}\ \alpha = \mathit{STRef}\ \mathit{RealWorld}\ \alpha$

The operators from Figure 5.1 can now be reused as *IORef* operators. Note that *IO* actions cannot be run via *runST*, since their σ-parameters are fixed to *RealWorld* and are therefore not arbitrary. This is a good thing, since otherwise, we could implement *unsafePerformIO* in terms of *runST*, which would mean that *runST* is unsafe.

5.2.2. Start Times as Type Parameters

We introduce type constructors for signal suffixes, which we name *DSuffix*, *CSuffix*, and *SSuffix*. Compared to signal type constructors, suffix type constructors have an additional parameter, usually called τ. The τ-parameter is a phantom parameter

that denotes the start time of the suffix. It is analogous to the σ-parameters of *ST* and *STRef*.

The suffix arguments and the result of a suffix combinator usually have the same start time. The only exception is the *switch* combinator. We enforce equality of start times the same way equality of variable sets is enforced for the basic *STRef* operators. For example, the type of *scanl* is

$$(\beta \to \alpha \to \beta) \to \beta \to \mathit{DSuffix}\ \tau\ \alpha \to \mathit{SSuffix}\ \tau\ \beta \ ,$$

and the type of *union* is

$$\mathit{DSuffix}\ \tau\ \alpha \to \mathit{DSuffix}\ \tau\ \alpha \to \mathit{DSuffix}\ \tau\ \alpha \ .$$

As noted in Section 5.1, we can only consume signals, that is, signal suffixes that start at t_0. Furthermore, producers only yield suffixes that start at t_0, since production happens at the start of the program only. We want to express this via the type system. For that, we introduce a type *Start* that denotes the start time of the program:

data *Start*

Start is analogous to *RealWorld* in that it is a concrete type for the otherwise fully polymorphic start time parameters. Using *Start*, we can define signal type constructors based on suffix type constructors in the same way we can define *IO* based on *ST*:

type *DSignal* α = *DSuffix Start* α
type *CSignal* α = *CSuffix Start* α
type *SSignal* α = *SSuffix Start* α

If our FRP system provides the functions *produce* and *consume*, the types of these functions continue to use signal type constructors, so that they only deal with suffixes that start at t_0.

5.3. Start Time Consistency and Switching

Switching in the context of start time consistency is a non-trivial issue. The reason is that switching deals with suffixes of different start times, which makes static guarantees about start times difficult. We solve this conflict in this section.

5.3.1. Security through Impredicativity

The *switch* combinator we defined in Subsection 3.2.5 has the type *SSignal* $(S\ \alpha) \to S\ \alpha$ for any signal type constructor S. Let us assume that there is a class *Signal*, whose instances are *DSignal*, *CSignal*, and *SSignal*. Then we can give a type to the generic combinator *switch*:

$$switch :: (Signal\ \sigma) \Rightarrow SSignal\ (\sigma\ \alpha) \to \sigma\ \alpha$$

We want to derive the type of a corresponding *switch* combinator for suffixes from the above type. So we have to care about start time parameters. It is clear that the result suffix of *switch* starts when the argument suffix starts. So the argument and the result type have to use the same type variable for their start time parameters. The suffixes that we switch to have different start times. However, they must be of a common type, which is used as the second parameter of *SSuffix* in the type of *switch*. We can solve this problem by using universal quantification.

For each suffix type constructor S, a value of a type $\forall\tau.S\ \tau\ \alpha$ describes a "start-time-agnostic signal suffix". An example of such a suffix is *empty*. It has the type $DSuffix\ \tau\ \alpha$, which is equivalent to $\forall\tau.DSuffix\ \tau\ \alpha$. This reflects the fact that there is an empty suffix for every start time. If we switch only to suffixes whose start time parameter is universally quantified, switching is start time consistent. Assuming a class *Suffix* with instances *DSuffix*, *CSuffix*, and *SSuffix*, we give *switch* the type

$$(Suffix\ \sigma) \Rightarrow SSuffix\ \tau\ (\forall\tau'.\sigma\ \tau'\ \alpha) \to \sigma\ \tau\ \alpha\ .$$

Note that this type uses more than ordinary higher-rank polymorphism, since the universally quantified type is the parameter of an ordinary data type. We need support for impredicative polymorphism [31] to use such types.

Alas, a *switch* combinator of the above type is secure, but essentially useless, since it only allows us to switch to "boring" suffixes. The only suffixes with arbitrary start time are the empty discrete suffix and all constant continuous and segmented suffixes. The reason is that all suffixes from producers have the fixed start time t_0. Suffixes inherited from them also inherit their start time. The only way to construct a suffix with arbitrary start time is to use suffix combinators only. However, the only combinators that yield suffixes without using existing suffixes are *empty* and *pure*. From empty and constant suffixes, the other combinators can only construct empty and constant suffixes again.

Even if it would be possible to switch to more interesting suffixes, we would have the problem that the only such suffixes start at t_0. What is missing is a way to age suffixes, that is, to trim them at their front such that they start at a later time. So we could only "switch" to a non-trivial suffix at the start of the program, but not later, which would not really make sense.

5.3.2. Suffix Functions to the Rescue

Our solution is to not switch between suffixes, but between functions whose types have the form

$$\sigma_1\ \tau\ \alpha_1 \to \ldots \to \sigma_n\ \tau\ \alpha_n \to \sigma\ \tau\ \alpha$$

where σ_1 through σ_n and σ are signal suffix type constructors. Because such functions can have different arities, there is no most general type for *switch* now.

$$
\begin{aligned}
&[\![switch]\!]\ (f_0, [\,]) &&= f_0\\
&[\![switch]\!]\ (f_0, (\Delta t, f) : \ddot{f}) &&= f' \textbf{ where}\\
&\quad f'\ s_1\ \ldots\ s_n &&= hop\ (f_0\ s_1\ \ldots\ s_n)\\
&&&\quad \Delta t\\
&&&\quad ([\![switch]\!]\ (f : \ddot{f})\ (age\ \Delta t\ s_1)\ \ldots\ (age\ \Delta t\ s_n))
\end{aligned}
$$

Figure 5.2.: Semantics of function switching

We present a solution to this problem in Subsection 5.3.3. For now, we pretend that *switch* has every type of the form

$$
\begin{aligned}
&(Suffix\ \sigma_1, \ldots, Suffix\ \sigma_n, Suffix\ \sigma) \Rightarrow\\
&SSuffix\ \tau\ (\forall \tau'.\sigma_1\ \tau'\ \alpha_1 \to \ldots \to \sigma_n\ \tau'\ \alpha_n \to \sigma\ \tau'\ \alpha) \to\\
&(\sigma_1\ \tau\ \alpha_1 \to \ldots \to \sigma_n\ \tau\ \alpha_n \to \sigma\ \tau\ \alpha)\ .
\end{aligned}
$$

To form the result of an expression $switch\ \bar{f}\ s_1\ \ldots\ s_n$, we split s_1 through s_n at the update times of $\bar{f}$. For each segment of $\bar{f}$, we compose the corresponding n slices of s_1 through s_n via the then current value of $\bar{f}$. We glue the resulting pieces together to get the final result. A formal definition of function switching is given in Figure 5.2.

A function that we switch to can only use its arguments and empty and constant suffixes to construct its result. Otherwise, our use of universal quantification would render it type-incorrect. So any "interesting" suffix that should be used in switching must be explicitely passed to *switch* as an additional argument. The *switch* combinator then guarantees that it is properly aged before it is switched to. So we have a switching combinator that allows us to switch to non-trivial suffixes and respects start time consistency at the same time. Furthermore, we have a way to age signal suffixes.

Let us now look at function switching in action. In Chapter 2, we presented the network monitor application. In the source code of this application, we defined the signal $\bar{v}$ along with a helper signal $\bar{\bar{v}}$ as follows:

$$
\begin{aligned}
&\bar{\bar{v}} = fmap\ (\lambda k \to \textbf{case}\ k\ \textbf{of}\ In \to \bar{v}_{In}; Out \to \bar{v}_{Out}; All \to \bar{v}_{All})\ \bar{k}\\
&\bar{v} = switch\ \bar{\bar{v}}
\end{aligned}
$$

Using function switching, we can define $\bar{v}$ in the following way, using a helper suffix $\bar{f}$ of functions:

$$
\begin{aligned}
&\bar{f} = fmap\ (\lambda k \to \textbf{case}\ k\ \textbf{of}\\
&\qquad\qquad In \to \lambda \bar{v}'_{In}\ \_\ \ \_ \to \bar{v}'_{In}\\
&\qquad\qquad Out \to \lambda \_\ \bar{v}'_{Out}\ \_ \to \bar{v}'_{Out}\\
&\qquad\qquad All \to \lambda \_\ \_\ \bar{v}'_{All} \to \bar{v}'_{All})\\
&\qquad \bar{k}\\
&\bar{v} = switch\ \bar{f}\ \bar{v}_{In}\ \bar{v}_{Out}\ \bar{v}_{All}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{data } \mathit{SuffixFun}\ \tau\ \varphi \textbf{ where}\\
&\quad \mathit{OSF} :: (\mathit{Suffix}\ \sigma) \Rightarrow \sigma\ \tau\ \alpha \rightarrow \mathit{SuffixFun}\ \tau\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha)\\
&\quad \mathit{SSF}\ :: (\mathit{Suffix}\ \sigma) \Rightarrow (\sigma\ \tau\ \alpha \rightarrow \mathit{SuffixFun}\ \tau\ \varphi) \rightarrow \mathit{SuffixFun}\ \tau\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \varphi)\\
&\textbf{data } \varphi \mapsto \varphi'\\
&\textbf{data } (\sigma :: * \rightarrow * \rightarrow *)\ \text{`}\mathit{Of}\text{`}\ (\alpha :: *)
\end{aligned}
$$

Figure 5.3.: Definition of *SuffixFun*

5.3.3. A Generic Suffix Function Type

So far, we have assumed that *switch* can work with functions of arbitrary arity. This is not directly possible with Haskell's type system. We overcome this problem by defining a Generalized Algebraic Data Type (GADT) *SuffixFun* whose values represent all functions of the form

$$\sigma_1\ \tau\ \alpha_1 \rightarrow \ldots \rightarrow \sigma_n\ \tau\ \alpha_n \rightarrow \sigma\ \tau\ \alpha$$

where σ_1 through σ_n and σ are instances of *Suffix*.

The definition of *SuffixFun*, including declarations of helper types, is shown in Figure 5.3. The φ-parameter of *SuffixFun* is a phantom parameter, called the function shape. A proper function shape has the form

$$\sigma_1\ \text{`}\mathit{Of}\text{`}\ \alpha_1 \mapsto \ldots \mapsto \sigma_n\ \text{`}\mathit{Of}\text{`}\ \alpha_n \mapsto \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha$$

with the same restrictions on σ_1 through σ_n and σ as above. Types of the form $\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha$ with σ being an instance of *Suffix* are called suffix shapes. Shapes do not contain start time parameters, because the single start time parameter of *SuffixFun* is used for all argument types and the result type of the function. The data constructors *OSF* and *SSF* construct nullary and non-nullary functions, respectively.[3]

Using *SuffixFun*, the type of *switch* becomes

$$\mathit{SSuffix}\ \tau\ (\forall \tau'.\mathit{SuffixFun}\ \tau'\ \varphi) \rightarrow \mathit{SuffixFun}\ \tau\ \varphi\ .$$

Compared to the non-solution of Subsection 5.3.1, we have just replaced the suffix type σ by *SuffixFun* and removed its type class constraint.

[3] *O* and *S* stand for "zero" and "successor". *SF* is shorthand for *SuffixFun*.

6. Vistas

In Section 4.2, we discussed several push-based implementation approaches. The most ambitious one, presented in Subsections 4.2.5 and 4.2.6, was to implement FRP based on improving times. This approach has the following desirable features:

- Event values are memoized, so that they are not unnecessarily recomputed when a signal is used multiple times.
- Simultaneity of events can be detected. As a result, glitches can be avoided when composing segmented signals.

However, there are also some issues:

- The implementation of improving times involves a lot of tricky low-level code, whose correctness is not easy to see.
- It is not guaranteed that events are handled in the order they occur.

In this chapter, we develop a push-based implementation that has the above-mentioned advantages while avoiding the disadvantages. We implement generators actually, but by using the techniques described in the last chapter, we can provide support for signal suffixes instead of generators. The contents of this chapter are largely taken from the already mentioned earlier work of the author [13].

6.1. Implementation of Suffix Types

We derive the implementation of *SSuffix* directly from the definition of $[\![\mathit{Signal}]\!]$. Since we derive a suffix type constructor instead of a signal type constructor, we also have to care about start time parameters. The update suffix of a segmented suffix $\bar{x}$ starts when $\bar{x}$ starts. So a segmented suffix and its update suffix have to use the same start time parameter. This leads to the following definition of *SSuffix*:

type $\mathit{SSuffix}\ \tau\ \alpha = (\alpha, \mathit{DSuffix}\ \tau\ \alpha)$

Now, we develop an implementation of discrete signal suffixes. We have seen in Subsections 4.1.3 and 4.2.4 that some *DSignal* implementations do not allow for a correct implementation of *merge*. Therefore, we consider merging right from the start. As an example of merging let us form the union of the signals $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ from Chapter 2. Remember that $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ represent the sequences of incoming and outgoing network packets, respectively. The signal $\mathit{union}\ \ddot{p}_{In}\ \ddot{p}_{Out}$ denotes an interleaving of these two sequences.[1]

[1] Remember that all events from $\ddot{p}_{Out}$ are included into the union, since $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ cannot both have an event at the same time.

If we represent discrete signal suffixes by lists of time–value pairs, we typically cannot deliver information about this interleaving early enough, as we have explained in Subsection 4.1.3. In Subsections 4.2.5 and 4.2.6, we saw a way to solve this problem, which is based on an advanced time type. In this chapter, we take a completely different approach. We let the representation of *union* $\ddot{p}_{In}$ $\ddot{p}_{Out}$ cover all possible interleavings. It is the responsibility of consumers to decide which one of them is the correct one.

The signals $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ are constructed directly by producers. Therefore, they correspond to two external sources of events, which we call e_{In} and e_{Out}. The order in that e_{In} and e_{Out} emit events determines the interleaving of $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ that makes up their union. Either e_{In} or e_{Out} will fire first. For both cases, the representation of the union gives the first event value and the remainder of the signal. The representation of the remainder is structured like the representation of the complete signal. So it distinguishes two cases according to what source will fire second and provides an event value and a remainder for each case. And so on.

We call the representation of a discrete signal suffix a vista and define the corresponding type *Vista* as follows:

```
type Vista    α = Map EventSrc (Variant α)
type Variant α = (α, Vista α)
```

A vista is a finite map from event sources to variants. The event sources are called the triggers of the vista, and the set of all triggers is called the vista's trigger set. A variant is a pair of a next event value and a remainder vista. Based on the *Vista* type, we define *DSuffix* as follows:

```
type DSuffix τ α = Vista α
```

An event source is represented by a pair of an ID and a registration action:[2]

```
type EventSrc = (Unique, Reg ())
```

Event source IDs makes it possible to compare sources for equality. We can even use them to define a total order on sources, since *Unique* is an instance of *Ord*:

```
instance Ord EventSrc where
  compare (EventSrc q1 _) (EventSrc q2 _) = compare q1 q2
```

Having an order is necessary for using sources as map keys in vistas. The registration action of an event source makes it possible to listen for events emitted by the source. Handlers registered via it are called with the trivial value (). This is no problem, since event values are communicated through vistas.

A vista corresponds to a kind of Mealy machine whose underlying graph is a tree. Figure 6.1 shows the machine that represents *union* $\ddot{p}_{In}$ $\ddot{p}_{Out}$. Thus, a label $\frac{e}{x}$ means that the respective transition is taken if the event source e fires, and that

[2] The type *Unique* used for IDs is defined in the *Data.Unique* module.

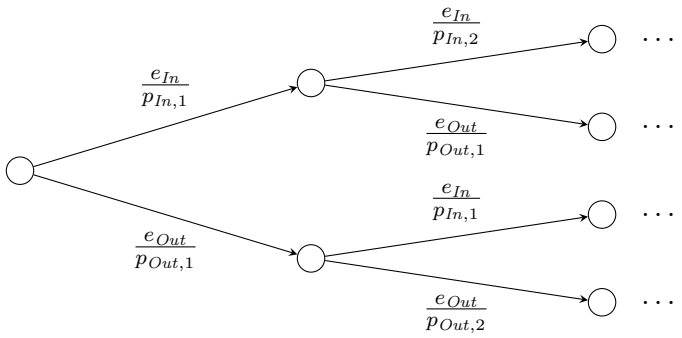

Figure 6.1.: Vista machine of *union* $\ddot{p}_{In}$ $\ddot{p}_{Out}$

there is an event with value x in this case. A variable $p_{k,i}$ denotes the value of the i-th event of the signal $\ddot{p}_k$. Compared to true Mealy machines, our vista machines differ in the following ways:

- The number of states can be infinite (albeit the number of successors of a state is always finite).
- If a source fires an event and there is no applicable transition for this source, the machine does nothing.

The above definition of vistas does not allow for implementing *filter*. We cannot drop events by removing transitions from the underlying machine. While this would prevent values from being output, it would also inhibit necessary state changes. Therefore, we change the vista concept such that outputs become optional. We only have to modify the definition of *Variant*:

type *Variant* $\alpha = (\mathit{Maybe}\ \alpha, \mathit{Vista}\ \alpha)$

6.2. Implementation of Suffix Combinators

Figure 6.2 shows vista-based implementations of *filter* and *scanl*. Both just recurse through the vista that represents the given discrete signal suffix. For this, they use the *fmap* operator on maps, which applies a function to the values that are assigned to keys. Note that we use identifiers with two dots also for vistas, since vistas are equivalent to discrete signal suffixes.

For implementing merging and switching, we introduce an auxiliary function *raceAndCont*, which is defined in Figure 6.3. A vista *raceAndCont* $l\ r\ b\ \ddot{x}_1\ \ddot{x}_2$ denotes a suffix that has no event occurrences until the time t where either $\ddot{x}_1$, $\ddot{x}_2$, or both have a first event occurrence. Let $\ddot{x}'_1$ and $\ddot{x}'_2$ denote the suffixes of

$$
\begin{aligned}
&\mathit{filter} :: (\alpha \to \mathit{Bool}) \to \mathit{DSuffix}\ \tau\ \alpha \to \mathit{DSuffix}\ \tau\ \alpha \\
&\mathit{filter}\ f = \mathit{fmap}\ (\lambda(\mathring{x}, \ddot{x}) \to (g\ \mathring{x}, \mathit{filter}\ f\ \ddot{x}))\ \textbf{where} \\
&\quad g\ \mathit{Nothing} \qquad\qquad = \mathit{Nothing} \\
&\quad g\ (\mathit{Just}\ x) \mid f\ x \qquad = \mathit{Just}\ x \\
&\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{Nothing} \\
&\mathit{scanl} :: (\beta \to \alpha \to \beta) \to \beta \to \mathit{DSuffix}\ \tau\ \alpha \to \mathit{SSuffix}\ \tau\ \beta \\
&\mathit{scanl}\ f\ y_0\ \ddot{x} = (y_0, a\ y_0\ \ddot{x})\ \textbf{where} \\
&\quad a\ y = \mathit{fmap}\ (\lambda(\mathring{x}, \ddot{x}) \to \textbf{case}\ \mathring{x}\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{Nothing} \to (\mathit{Nothing}, a\ y\ \ddot{x}) \\
&\qquad\qquad\qquad\qquad \mathit{Just}\ x \quad \to \textbf{let} \\
&\qquad\qquad\qquad\qquad\qquad\qquad y' = f\ y\ x \\
&\qquad\qquad\qquad\qquad\qquad \textbf{in}\ (\mathit{Just}\ y', a\ y'\ \ddot{x}))
\end{aligned}
$$

Figure 6.2.: Vista-based implementation of selected combinators

$\ddot{x}_1$ and $\ddot{x}_2$ that start at t. The behavior of *raceAndCont* $l\ r\ b\ \ddot{x}_1\ \ddot{x}_2$ from t onwards is described by a *Variant* value v that is defined as follows:

$$
v = \begin{cases}
l\ x_1 \quad\ \ddot{x}_1'\ \ddot{x}_2' & \text{if at } t,\ \ddot{x}_1 \text{ has an event with value } x_1, \text{ and } \ddot{x}_2 \text{ has no event} \\
r \quad\ x_2\ \ddot{x}_1'\ \ddot{x}_2' & \text{if at } t,\ \ddot{x}_2 \text{ has an event with value } x_2, \text{ and } \ddot{x}_1 \text{ has no event} \\
b\ x_1\ x_2\ \ddot{x}_1'\ \ddot{x}_2' & \text{if at } t,\ \ddot{x}_1 \text{ and } \ddot{x}_2 \text{ have events with values } x_1 \text{ and } x_2
\end{cases}
$$

The *raceAndCont* implementation uses a helper function h. Let x and x' be two vistas, and let T and T' be their trigger sets. The vista $h\ \ddot{x}\ \ddot{x}'$ denotes the same suffix as $\ddot{x}$, but its trigger set is $T \cup T'$ instead of T. When a source from $T' \setminus T$ fires, the vista machine of $h\ \ddot{x}\ \ddot{x}'$ performs a step without yielding an event value. It then continues like the machine of $\ddot{x}$. The identifier *union* that is used in the implementation of h denotes the union operator for maps. Since *union* is left-biased, $h\ \ddot{x}\ \ddot{x}'$ maps all sources from $T \cap T'$ to the corresponding variants from $\ddot{x}$.

Let T_1 and T_2 be the trigger sets of the *raceAndCont* arguments $\ddot{x}_1$ and $\ddot{x}_2$. The vistas $h\ \ddot{x}_1\ \ddot{x}_2$ and $h\ \ddot{x}_2\ \ddot{x}_1$ are equivalent to $\ddot{x}_1$ and $\ddot{x}_2$, respectively, but have the common trigger set $T_1 \cup T_2$. We combine each variant from $h\ \ddot{x}_1\ \ddot{x}_2$ with the one from $h\ \ddot{x}_2\ \ddot{x}_1$ that belongs to the same event source. The *unionWith* combinator that we use for this is the *unionWith* combinator for maps.

Figure 6.4 shows the implementation of *merge*. The vista machine of a suffix *merge* $l\ r\ b\ \ddot{x}_1\ \ddot{x}_2$ has to simulate the machines of $\ddot{x}_1$ and $\ddot{x}_2$ in parallel. So merging two suffixes essentially means calculating a kind of product automaton. This is easy because of the tree structure of vistas. We employ the *raceAndCont* function, which directly generates the transitions from the initial state to its successors. We embed calls to *merge* in the function arguments of *raceAndCont*, so that *merge* is applied recursively to subvistas.

$$
\begin{aligned}
&\mathit{raceAndCont} :: (\alpha \to \quad \mathit{Vista}\ \alpha \to \mathit{Vista}\ \beta \to \mathit{Variant}\ \gamma) \\
&\qquad\qquad\quad (\quad \beta \to \mathit{Vista}\ \alpha \to \mathit{Vista}\ \beta \to \mathit{Variant}\ \gamma) \\
&\qquad\qquad\quad (\alpha \to \beta \to \mathit{Vista}\ \alpha \to \mathit{Vista}\ \beta \to \mathit{Variant}\ \gamma) \\
&\qquad\qquad\quad (\mathit{Vista}\ \alpha \to \mathit{Vista}\ \beta \to \mathit{Vista}\ \gamma) \\
&\mathit{raceAndCont}\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2 = \mathit{unionWith}\ c\ (h\ \ddot{x}_1\ \ddot{x}_2)\ (h\ \ddot{x}_2\ \ddot{x}_1)\ \textbf{where} \\
&\quad c\ (\mathit{Nothing}, \ddot{x}_1)\ (\mathit{Nothing}, \ddot{x}_2) = (\mathit{Nothing}, \mathit{raceAndCont}\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2) \\
&\quad c\ (\mathit{Nothing}, \ddot{x}_1)\ (\mathit{Just}\ x_2,\ \ddot{x}_2) = r \quad\ x_2\ \ddot{x}_1\ \ddot{x}_2 \\
&\quad c\ (\mathit{Just}\ x_1,\ \ddot{x}_1)\ (\mathit{Nothing}, \ddot{x}_2) = l\ x_1 \quad\ \ddot{x}_1\ \ddot{x}_2 \\
&\quad c\ (\mathit{Just}\ x_1,\ \ddot{x}_1)\ (\mathit{Just}\ x_2,\ \ddot{x}_2) = b\ x_1\ x_2\ \ddot{x}_1\ \ddot{x}_2 \\
&\quad h :: \mathit{Vista}\ \delta \to \mathit{Vista}\ \varepsilon \to \mathit{Vista}\ \delta \\
&\quad h\ \ddot{x}\ \ddot{x}' = \mathit{union}\ \ddot{x}\ (\mathit{fmap}\ (\mathit{const}\ (\mathit{Nothing}, \ddot{x}))\ \ddot{x}')
\end{aligned}
$$

Figure 6.3.: Implementation of *raceAndCont*

$$
\begin{aligned}
&\mathit{merge} :: (\alpha \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad \to \\
&\qquad\quad (\beta \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad \to \\
&\qquad\quad (\alpha \to \beta \to \gamma) \qquad\qquad\qquad\qquad\qquad\qquad \to \\
&\qquad\quad (\mathit{DSuffix}\ \tau\ \alpha \to \mathit{DSuffix}\ \tau\ \beta \to \mathit{DSuffix}\ \tau\ \gamma) \\
&\mathit{merge}\ l\ r\ b = \mathit{raceAndCont}\ (\lambda x_1 \quad\ \ddot{x}_1\ \ddot{x}_2 \to (\mathit{Just}\ (l\ x_1), \quad\ \mathit{merge}\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2)) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\lambda \quad x_2\ \ddot{x}_1\ \ddot{x}_2 \to (\mathit{Just}\ (r\ x_1), \quad\ \mathit{merge}\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2)) \\
&\qquad\qquad\qquad\qquad\qquad\quad (\lambda x_1\ x_2\ \ddot{x}_1\ \ddot{x}_2 \to (\mathit{Just}\ (b\ x_1\ x_2), \mathit{merge}\ l\ r\ b\ \ddot{x}_1\ \ddot{x}_2))
\end{aligned}
$$

Figure 6.4.: Vista-based implementation of *merge*

```
data DSuffixFun φ where
  ODSF :: DSuffix τ α →                          DSuffixFun τ (DSuffix `Of` α)
  SDSF :: (DSuffix τ α → DSuffixFun τ φ) → DSuffixFun τ (DSuffix `Of` α ↦ φ)
unODSF :: DSuffixFun τ (DSuffix `Of` α) → DSuffix τ α
unODSF (ODSF ẍ) = ẍ
unSDSF :: DSuffixFun τ (DSuffix `Of` α ↦ φ) → DSuffix τ α → DSuffixFun τ φ
unSDSF (SDSF f) = f
```

Figure 6.5.: Definition of *DSuffixFun* and associated destructors

```
dSwitch :: SSuffix τ (∀τ'.DSuffixFun τ' φ) → DSuffixFun τ φ
dSwitch f̄@(ODSF _,_) = ODSF $ v_O (fmap unODSF f̄) where
  v_O :: SSuffix τ (Vista α) → Vista α
  v_O = uncurry $ raceAndCont (λx    ẍ  ⃛x' → (Just x,   v_O (ẍ,  ⃛x')))
                              (λ   ẍ' _ ⃛x' → (Nothing, v_O (ẍ', ⃛x')))
                              (λx ẍ' _ ⃛x' → (Just x,   v_O (ẍ', ⃛x')))
dSwitch f̄@(SDSF _,_) = SDSF $ v_S (fmap unSDSF f̄) where
  v_S :: SSuffix τ (∀τ'.Vista α → DSuffixFun τ' φ) → Vista α → DSuffixFun τ φ
  v_S (f_0, f̈) ẍ = dSwitch (f_0 ẍ, a f̈ ẍ)
  a :: Vista (∀τ'.Vista α → DSuffixFun τ' φ) → Vista α → Vista (DSuffixFun τ φ)
  a = raceAndCont (λf    f̈ ẍ → (Just (f ẍ), a f̈ ẍ)
                  (λ   _ f̈ ẍ → (Nothing,    a f̈ ẍ)
                  (λf _ f̈ ẍ → (Just (f ẍ), a f̈ ẍ)
```

Figure 6.6.: Vista-based implementation of *dSwitch*

The implementation of *switch* is rather complex. The reason is that the *SuffixFun* type is very general, so that *switch* has to be able to deal with a multitude of different cases. Therefore, we only discuss a restricted switching combinator *dSwitch* that only deals with discrete suffixes. Figure 6.5 defines the type of all discrete signal suffix functions along with two destructor functions. The implementation of *dSwitch* is shown in Figure 6.6. It distinguishes between nullary and and non-nullary signal suffix functions and converts between signal suffix functions and suffixes or ordinary functions, respectively. The actual work is delegated to the helper functions v_O and v_S.

The v_O function turns a segmented signal suffix of vistas into a single vista by successively switching to the vistas from the segmented suffix. It combines the initial vista with the update vista using *raceAndCont*. The events of the initial vista are taken over to the result until a vista update occurs. Then, the new vista takes the place of the initial vista, and v_O continues as before.

```
produce :: Reg α → IO (DSignal α)
produce r = do
              c ← newChan
              r (writeChan c)
              q ← newUnique
              let
                e = (q, λh → r (λ_ → h ()))
              xs ← getChanContents c
              return (foldr (λx ẍ → singleton e (Just x, ẍ)) ⊥ xs)
```

Figure 6.7.: Vista-based implementation of *produce*

The v_S function switches between ordinary functions from vistas to signal suffix functions. It first turns the segmented signal suffix of functions into a segmented signal suffix of function results by applying the functions to those suffixes of the given vista that start when the respective functions come into effect. Afterwards, v_S calls *dSwitch* recursively to switch between the function results.

The helper function a generates the update vista of the function result suffix. The vistas $\ddot{x}$ it deals with internally are aged versions of the argument vista of v_S. Ageing is done via *raceAndCont* as time progresses. So all parts of the argument vista that will not be needed anymore can be garbage-collected, and at the time of switching, we have the aged suffix right available. As a result, we have avoided the space and time efficiency problems that we mentioned at the end of Subsection 4.2.5. This has only been possible because the type of the switching combinator forces us to specify all suffixes that we want to switch to later as function arguments. So these suffixes are known right from the start, which is necessary to age them right from the start.

6.3. Implementation of Production and Consumption

The code for *produce* is shown in Figure 6.7. It uses channel support as provided by the module *Control.Concurrent.Chan* [23]. Channels are actually intended for communication between concurrent processes. However, *produce* does not employ concurrency at all. It uses channels, since they make it possible to turn a sequence of values produced by I/O actions into a lazy list.

To produce a discrete signal $\ddot{x}$ from a registration action r, we first create a channel. Every event that is emitted by the event source that $\ddot{x}$ mirrors has to be put into that channel. We achieve this by registering an appropriate handler via r. Afterwards, we create an *EventSrc* value e that represents the observed event source. It consists of a fresh ID and a registration action that is equivalent to r with the exception that it makes handlers called with () instead of event values.

```
consume :: DSignal α → Reg α
consume ẍ = λh → do
                    u⃗ ← newIORef ⊥
                    let
                      w ẍ = do
                              us ← mapM a (assocs ẍ)
                              writeIORef u⃗ (sequence_ us)
                      a ((_, r), (x̊, ẍ)) = r (λ_ → do
                                                    when (isJust x̊)
                                                         (h (fromJust x̊))
                                                    join (readIORef u⃗)
                                                    w ẍ)
                    w ẍ
                    return (join (readIORef u⃗))
```

Figure 6.8.: Vista-based implementation of *consume*

We use *getChanContents* to get a lazy list of all values that will be put into the channel. We transform this list into a discrete signal that yields a value every time e fires.

The implementation of *consume* is given in Figure 6.8. To register a handler h with a signal $\ddot{x}$, we first create a mutable variable. The purpose of this variable is to always hold an I/O action that unregisters h. The major work is done by the helper function w. This function takes a vista, registers a handler with every trigger of the vista, and stores an I/O action that undoes all these registrations in the mutable variable. The handler that is registered with a trigger e fetches the variant that is assigned to e. If the variant contains an event value, the respective event is handled. Afterwards, all registrations are undone, and w is called with the remainder vista. We get the ball rolling by applying w to the consumed signal $\ddot{x}$. Finally, we return an I/O action that just executes the I/O action currently stored in the mutable variable.

The *consume* function walks through the vista on the path that describes the actual behavior of the consumed signal. Only this path is evaluated by *consume*. All other parts of the vista describe possibilities that do not arise. They are left unevaluated and can be garbage-collected as soon as it becomes clear that they do not describe reality.

6.4. Performance Comparison

We compare the performance of the current Grapefruit version, which uses vistas, with the performance of the version that directly preceded the introduction of

vistas. The implementation ideas behind this pre-vista version are described in an earlier work by the author [11]. A specific problem of the pre-vista approach is that event values are computed multiple times if a signal suffix is consumed more than once, or if it is used more than once in the calculation of another suffix. This is basically the problem that we discussed at the beginning of Subsection 4.2.3.

Performance is especially crucial when events occur very frequently. An example of such a situation is real-time sound synthesis, which we discuss here. We generate a signal that represents a sequence of timer ticks. From this signal, we calculate audio signals. Each audio signal assigns a block of multiple consecutive samples to every tick. We create a sound using FM synthesis and fade it out by multiplying it with an exponential function. The resulting signal is processed by a number of consumers. Each consumer iterates through all samples and evaluates those that are still unevaluated. We variegate the number of consumers between 1 and 10.

To ease measurements, we do not produce actual timer ticks. Instead, we generate a tick at the beginning of the simulation and every time the reaction to a previous tick has been finished. That way, the program never becomes idle. Since a sampling rate of 96 kHz and a tick frequency of about 1 kHz are typical for audio processing, we fix the block size to 100 samples. In each simulation run, we process 10 000 blocks.

We perform our measurements on an Intel Pentium M processor with a clock frequency of 600 MHz. We compile the code using version 6.10.4 of the Glasgow Haskell Compiler with optimizations turned on. For each of the two Grapefruit versions and each number of consumers, we run the simulation five times and form the mean of the total CPU times.

The results of our measurements are shown in Figure 6.9. We can see that the new implementation already pays off in the case of only two consumers. Note that it is not unusual for a signal to be used more than two times. For example, a sound synthesizer might compose a signal with a variation of itself to achieve a phaser effect and feed the resulting signal into four different audio channels. This would result in the original signal being used eight times.

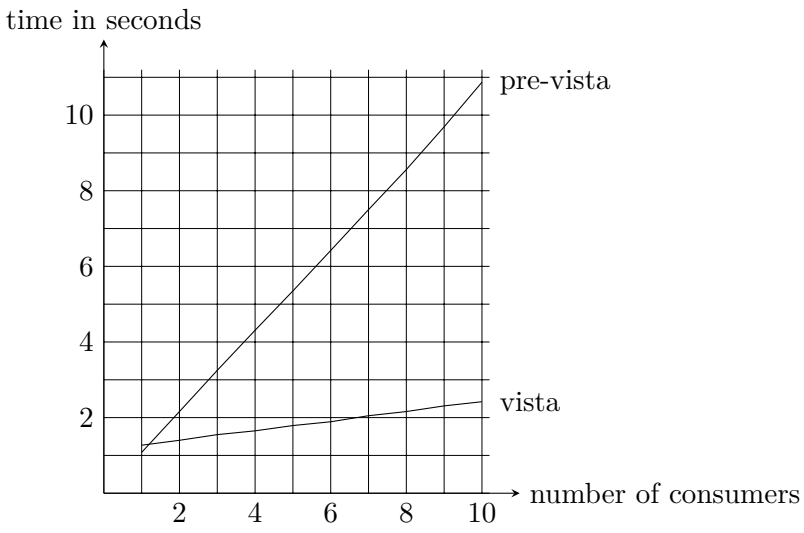

Figure 6.9.: Performance comparison between vista and pre-vista Grapefruit

7. A Generic Record System

Functional Reactive Programming benefits from a record system that allows for defining generic record combinators in a type-safe fashion. When this became apparent during the development of Grapefruit, we could not find any such system. Therefore, we developed one on our own. Our record system is implemented as a library and consists of the packages *records*, *kinds*, and *type-functions*[1]. We describe its core ideas in this chapter and an advanced feature in the next one.

In Section 7.1, we motivate the ideas presented in this chapter. We describe a simple record system in Section 7.2. This serves as a starting point for our developments. Sections 7.3 and 7.4 present the novel concepts of record type families and record scheme induction, respectively. In Section 7.5, we implement a generic record conversion operator and show how record pattern matching can be done based on this operator. The contents of this chapter are largely taken from an earlier work of the author [12], with permission from the copyright holder, as explained in Appendix A.

7.1. Motivation

In real world applications, we often encounter groups of related signals. As an example, let us look at the network monitor application from Chapter 2 again. The signals $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ both give information about network traffic. So it is sensible to bundle them into a single value. Figure 7.1 defines a type *Traffic* for this purpose. This type uses Haskell's simple record support for giving names to the individual signals.

In Subsection 4.2.1, we showed how to implement *getInTraffic* in push-based FRP systems. We just apply *produce* to a registration action *regInPacketHandler*. Assuming we have a registration action *regOutPacketHandler*, we can implement

[1] These packages are available via `http://hackage.haskell.org/packages/archive/pkg-list.html`.

```
data Traffic = Traffic {
                 inTraffic  :: DSignal Packet,
                 outTraffic :: DSignal Packet
               }
```

Figure 7.1.: Definition of *Traffic* using Haskell's built-in record system

```
getTraffic :: IO Traffic
getTraffic = do
               p̈In  ← produce regInPacketHandler
               p̈Out ← produce regOutPacketHandler
               return $ Traffic {
                          inTraffic  = p̈In,
                          outTraffic = p̈Out
                        }
```

Figure 7.2.: Implementation of *getTraffic* using Haskell's built-in record system

```
data WindowActions = WindowActions {
                       close           :: DSignal ()
                       minimize        :: DSignal ()
                       maximize        :: DSignal ()
                       moveToWorkspace :: DSignal Integer
                     }
```

Figure 7.3.: Definition of *WindowActions* using Haskell's built-in record system

getOutTraffic in an analogous way. Now, we use both registration actions to implement an I/O action *getTraffic* that yields a *Traffic* value. The code is shown in Figure 7.2.

Say we want to produce a record of signals that provide information about user actions on a GUI window. In particular, we want to observe presses on the window's close, minimize, and maximize buttons as well as requests for moving the window to a different workspace of the desktop. Figure 7.3 defines a suitable record type. We want to provide a function *getWindowActions* that turns a window identifier into an I/O action that yields the corresponding *WindowActions* value. A typical implementation of *getWindowActions* looks very similar to the one of *getTraffic*. It calls *produce* with different registration actions and creates a record out of the results.

Implementing more functions similar to *getTraffic* and *getWindowActions* can quickly get cumbersome, since we have to use the same code pattern over and over again. So we want to come up with a generic combinator *multiProduce* that can produce records of discrete signals from arbitrary records of registration actions. This poses the following challenges:

- The combinator has to accept all record types whose field types have the form *Reg* α as input types, but no others.
- The combinator has to accept all record types whose field types have the form *DSignal* α as output types, but no others.

```
data InTraffic         = InTraffic
data OutTraffic        = OutTraffic
data Close             = Close
data Minimize          = Minimize
data Maximize          = Maximize
data MoveToWorkspace = MoveToWorkspace
```

Figure 7.4.: Definition of example name types

- The type of the combinator has to ensure that there is a one-to-one correspondence between the fields of the input record and the fields of the output record, and that the types of corresponding fields use the same type parameter for *Reg* and *DSignal*, respectively.

- The combinator has to iterate through all fields of the input record to produce all fields of the output record. It is not enough to deal with a fixed number of fields using statically known field names.

Haskell's record system is by far not powerful enough to solve these problems. Some advanced record systems have been proposed [9, 14], but they are also not capable of dealing with the above challenges. The reason is that they only provide operations that work with single fields instead of whole records. On the other hand, we can use the wide variety of type system extensions supported by the Glasgow Haskell Compiler (GHC) to implement our own record system as a library. This is the route we follow here. Our work is based on ideas from HList [15]. This is a Haskell library for statically-typed heterogenous lists, that is, lists whose elements may have different types. HList uses heterogenous lists of name–value pairs to represent records.

7.2. A Simple Record Library

We first show a simple implementation of records as lists of name-value pairs. The types of these lists specify the names of the record fields along with the types of the corresponding values. We develop our implementation bottom-up, starting with the representation of field names.

We have to represent names both at the type level and at the value level. For each name, we declare a nullary type constructor with a single nullary data constructor that uses the same identifier as the type constructor. So we declare the names used in the introductory examples as shown in Figure 7.4.

A record field is a pair of a name and a value. So we could use ordinary pairs to represent fields. We do not do that, however, because it would lead to shabby

$$
\begin{array}{lll}
\textbf{type } \mathit{Traffic} & = X :\& \mathit{InTraffic} & ::: \mathit{DSignal\ Packet} \\
 & :\& \mathit{OutTraffic} & ::: \mathit{DSignal\ Packet} \\
\textbf{type } \mathit{WindowActions} & = X :\& \mathit{Close} & ::: \mathit{DSingal}\ () \\
 & :\& \mathit{Minimize} & ::: \mathit{DSignal}\ () \\
 & :\& \mathit{Maximize} & ::: \mathit{DSignal}\ () \\
 & :\& \mathit{MoveToWorkspace} & ::: \mathit{DSignal\ Integer}
\end{array}
$$

Figure 7.5.: Library-based definition of signal record types

syntax.[2] Instead, we declare a special field type:

$$\textbf{data}\ \nu ::: \alpha = \nu := \alpha$$

The operator symbol ::: was chosen because it is similar to the special symbol ::, which stands for "has type".

We define records as heterogenous lists of fields. We introduce two constructors, one for the empty record and one for record extension:

$$
\begin{aligned}
&\textbf{data}\ X \quad\ = X \\
&\textbf{data}\ \rho :\& \varphi = \rho :\& \varphi
\end{aligned}
$$

While we could use the unit type () and the pair type (,) instead of X and (:&), we choose to introduce new types for similar reasons as we did for record fields. Note that (:&) is a "snoc", not a "cons", that is, new fields are appended, not prepended. In the following, we assume that :& is left-associative and of lower priority than ::: and :=.

The types *Traffic* and *WindowActions* can now be defined as shown in Figure 7.5. The traffic record consisting of the signals $\ddot{p}_{In}$ and $\ddot{p}_{Out}$ can be written

$$X :\& \mathit{InTraffic} := \ddot{p}_{In} :\& \mathit{OutTraffic} := \ddot{p}_{Out}\ .$$

Figure 7.6 defines types of registration action records that correspond to *Traffic* and *WindowActions*. Note that *TrafficRegs* and *WindowActionRegs* use the same field names as the corresponding signal record types. That way, *multiProduce* knows the names of the fields that shall be produced.

Records according to the above definition are not name-to-value maps. There are the following issues:

- A name may occur multiple times in the same record. Since this can even be an advantage [18], we retain this property.

[2] The actual implementation in the *records* package also makes field names strict, which is another reason for not using the ordinary pair type.

```
type TrafficRegs         = X :& InTraffic          ::: Reg Packet
                           :& OutTraffic           ::: Reg Packet
type WindowActionsRegs = X :& Close              ::: Reg ()
                           :& Minimize           ::: Reg ()
                           :& Maximize           ::: Reg ()
                           :& MoveToWorkspace    ::: Reg Integer
```

Figure 7.6.: Library-based definition of registration action record types

```
class MultiProduce ρ ϱ
instance MultiProduce X X
instance (MultiProduce ρ ϱ) ⇒
         MultiProduce (ρ :& ν ::: Reg α) (ϱ :& ν ::: DSignal α)
```

Figure 7.7.: Definition of class *MultiProduce*

- The fields of a record are ordered. Having an order is good for fields with the same name since it allows us to distinguish such fields by relative position. However, it introduces redundancy otherwise. We show how we can overcome the redundancy problem in Section 7.5.

In the next section, we refine our definition of record types by introducing the concept of record type families. This makes it possible to specify type constraints like the ones for the *multiProduce* combinator that we outlined in Section 7.1. Section 7.4 presents a record-related fold operator. This operator allows us to finally implement *multiProduce*.

7.3. Record Type Families

Let us now develop a type for *multiProduce* that encodes the constraints on the input and the output type we have identifed in Section 7.1. As a first approach, we introduce a class *MultiProduce* whose instances are all pairs of a registration action record type and its corresponding signal record type. Figure 7.7 shows the definition of such a class, including its instance declarations. Now, *multiProduce* can be given the type

$$(\mathit{MultiProduce}\ \rho\ \varrho) \Rightarrow \rho \to \mathit{IO}\ \varrho\ .$$

This approach is similar to what is done in HList.

The downside of this approach is, that one gets many classes that are related to each other without their relationships being known to the type checker. For example, we can define a class *MultiConsume* as an analog to *MultiProduce*. The

$$\begin{aligned}
&\textbf{data } X && \sigma = X\\
&\textbf{data } (\rho :\& \varphi) && \sigma = \rho\ \sigma :\& \varphi\ \sigma\\
&\textbf{data } (\nu ::: \varsigma) && \sigma = \nu := \sigma\ \varsigma
\end{aligned}$$

Figure 7.8.: Definition of record schemes

types of the records that can be consumed have the same structure as the types of the records that can be produced. However, the type checker is not able to see this. This can result in large contexts that contain redundant information. Record type families provide a way out of this problem.

7.3.1. Record Type Family Essentials

The input and the output record of *multiProduce* use the same field names. For every name ν that is used, there is a type α_ν such that the input record assigns a value of type *Reg* α_ν to ν, while the output record assigns a value of type *DSignal* α_ν to ν. There are no restrictions on the type α_ν itself. So we can identify the valid pairs of an input record type and an output record type by mappings from names ν to types α_ν. We can transform such mappings into the corresponding input record types and output record types. Record type families allow us to perform such transformations on the fly.

A record type family is characterized by a record scheme. A record scheme is a list of pairs, each consisting of a name and a so-called sort. A sort is a type, so record schemes have the same structure as our record types from the previous section. However, sorts are not used as value types directly. We build a record type by combining a record scheme with a type-level function, which is called the style of the record type. The record style is applied to all sorts of the scheme to generate the value types of the respective fields. By coupling the same scheme with different styles, we get a family of related record types.

We modify the declarations of X, $(:\&)$, and $(:::)$ such that types of the form

$$X :\& \nu_1 ::: \varsigma_1 :\& \ldots :\& \nu_n ::: \varsigma_n$$

denote record schemes with names ν_1 through ν_n and sorts ς_1 through ς_n. Applying such a type to a style yields the respective record type. The new declarations are given in Figure 7.8. Figure 7.9 defines the class *Record* of all record schemes.

Now, we want to use record type families to give a type to *multiProduce*. We generate the type of the input record and the type of the output record from the same scheme. This scheme uses the types α_ν mentioned above as its sorts. So the style of the input record has to be *Reg*, and the style of the output record has to be *DSignal*. As a result, *multiProduce* has the type

$$(\mathit{Record}\ \rho) \Rightarrow \rho\ \mathit{Reg} \to \mathit{IO}\ (\rho\ \mathit{DSignal})\ .$$

class	*Record* ρ
instance	*Record* X
instance (*Record* ρ) $\Rightarrow$	*Record* $(\rho :\& \nu ::: \varsigma)$

Figure 7.9.: Definition of class *Record* without methods

7.3.2. Type-Level Abstractions

Let us look at a more complicated example. We want to develop a function *multiScanl1* that receives a record of accumulation functions and a record of discrete signals and composes them field-wise using *scanl1*. The style for the accumulation function record maps each type α to the type $\alpha \to \alpha \to \alpha$. Alas, there is no type constructor or partial application of a type constructor that performs this mapping. Let us assume we had type-level abstractions available such that $\lambda\alpha \to \tau'$ denotes a type-level function that maps each type τ of kind $*$ to $\tau'[\tau/\alpha]$. Now, we can denote the style of the accumulation function record by $\lambda\alpha \to (\alpha \to \alpha \to \alpha)$.

Unfortunately, Haskell does not support type-level abstractions. However, we can emulate them by using defunctionalization [24] at the type level. We represent type-level functions by ordinary types and introduce a type synonym family [27] that describes type-level function application:

type family *App* φ α

Instances of *App* have to be defined such that for each type-level function F with representation φ and each argument type α, the type *App* φ α equals F α. For each type-level abstraction $\lambda\alpha \to \tau'$ with free variables β_1 through β_n, we introduce an n-ary type constructor Λ whose arguments have the same kinds as β_1 through β_n. In addition, we add the following instance declaration for *App*:

type instance *App* $(\Lambda\ \beta_1\ \ldots\ \beta_n)\ \alpha = \tau'$

Then, the type $\Lambda\ \beta_1\ \ldots\ \beta_n$ represents the type-level function $\lambda\alpha \to \tau'$.

We have to modify the record-related types such that they use representations of type-level functions instead of the functions themselves. We can keep the data declarations of X and $(:\&)$, but have to change the data declaration of $(:::)$ to use *App*. The complete definition of record schemes with support for type-level abstractions is presented in Figure 7.10.

We will now formulate the type of *multiScanl1* using emulation of type-level abstractions. We introduce a type *AccuStyle* for representing the style of accumulation function records and an *App* instance declaration for this type, which mirrors the abstraction $\lambda\alpha \to (\alpha \to \alpha \to \alpha)$:

data *AccuStyle*
type instance *App AccuStyle* $\alpha = \alpha \to \alpha \to \alpha$

$$\begin{aligned}
&\textbf{data } X \qquad\quad \sigma = X \\
&\textbf{data } (\rho :\& \varphi)\ \sigma = \rho\ \sigma :\& \varphi\ \sigma \\
&\textbf{data } (\nu ::: \varsigma)\ \ \sigma = \nu := \mathit{App}\ \sigma\ \varsigma
\end{aligned}$$

Figure 7.10.: Definition of record schemes with support for type-level abstractions

Furthermore, we introduce a type constructor *TypeStyle* and an accompanying *App* instance such that for each type τ of kind $* \to *$, *TypeStyle* τ represents τ, which is equivalent to $\lambda\alpha \to \tau\ \alpha$:

$$\begin{aligned}
&\textbf{data } \mathit{TypeStyle}\ (\tau :: * \to *) \\
&\textbf{type instance } \mathit{App}\ (\mathit{TypeStyle}\ \tau)\ \alpha = \tau\ \alpha
\end{aligned}$$

Now, the type of *multiScanl1* is

$$(\mathit{Record}\ \rho) \Rightarrow \rho\ \mathit{AccuStyle} \to \rho\ (\mathit{TypeStyle}\ \mathit{DSignal}) \to \rho\ (\mathit{TypeStyle}\ \mathit{DSignal})\ .$$

So far, *multiScanl1* only works with records of signals. Now, we generalize it such that it also works with records of signal suffixes that start after t_0. However, we require that all suffixes within a record start at the same time. The type of *multiScanl1* now contains a type variable τ that denotes the start time of the input and output suffixes. The style of the suffix records is $\lambda\alpha \to \mathit{DSuffix}\ \tau\ \alpha$, so it contains τ as a free variable. Therefore, the representation of this style has a parameter, which denotes the start time of the suffixes. It is defined as follows:

$$\begin{aligned}
&\textbf{data } \mathit{DSuffixStyle}\ \tau \\
&\textbf{type instance } \mathit{App}\ (\mathit{DSuffixStyle}\ \tau)\ \alpha = \mathit{DSuffix}\ \tau\ \alpha
\end{aligned}$$

The type of *multiScanl1* is now

$$(\mathit{Record}\ \rho) \Rightarrow \rho\ \mathit{AccuStyle} \to \rho\ (\mathit{DSuffixStyle}\ \tau) \to \rho\ (\mathit{DSuffixStyle}\ \tau)\ .$$

7.4. Record Scheme Induction

Let us now try to implement the *multiProduce* function for producing records of discrete signals. We can implement *multiProduce* using induction on its record scheme parameter. Since record schemes are types, not values, we do not use pattern matching to distinguish between empty and non-empty record schemes. Instead, we make *multiProduce* a method of the *Record* class and provide a method declaration for each of the two cases as part of the respective instance declaration. Figure 7.11 shows the complete code.

Of course, we also want to use record scheme induction to implement other combinators than *multiProduce*. For each of them, we have three different options of implementing it:

```
class Record ρ where
  multiProduce :: ρ (TypeStyle Reg) → IO (ρ (TypeStyle DSignal))
instance Record X where
  multiProduce X = return X
instance (Record ρ) ⇒ Record (ρ :& ν ::: ς) where
  multiProduce (r̂ :& n := r) = do
                                 x̂̂ ← multiProduce r̂
                                 ẍ ← produce r
                                 return (x̂̂ :& n := ẍ)
```

Figure 7.11.: Definition of class *Record* with method *multiProduce*

1. We can add a new class that has the same instances as *Record* and contains the combinator as its method.
2. We can implement the combinator as a new method of the *Record* class or of one of the classes created according to option 1.
3. We can implement the combinator as an ordinary function.

If we use option 1, we end up with multiple classes that have the same instances, but the type checker cannot see that their instances are the same. So if we use multiple record combinators in a single expression, the type of the expression may contain lots of different class assertions that all mean the same thing. Therefore, we drop option 1.

Now, all inductively defined record combinators must be either methods of *Record* or ordinary functions. Once a class is declared, its set of methods is fixed. So we have to decide once and for all which combinators shall be implemented as methods of *Record* at the time we declare *Record*. Only these combinators can use record scheme induction directly by having different declarations for the two instance declarations of *Record*. All other combinators can use induction only indirectly by applying the methods of *Record*.

7.4.1. Folding Record Schemes

We declare a single method of *Record* that captures induction over record schemes in full generality. That way, every inductively defined record combinator can be implemented as an ordinary function that uses that method. Induction principles are represented by fold operators. So what we want is a fold over record schemes. We remove the *multiProduce* method from the *Record* class and add a method *fold*. Figure 7.12 shows the resulting definition of *Record*.

We can produce an inductively defined record combinator by applying *fold* to an X-alternative and a (:&)-alternative. These alternatives describe how specializations

```
class Record ρ where
  fold :: θ X → (∀ρ ν ς.(Record ρ) ⇒ θ ρ → θ (ρ :& ν ::: ς)) → θ ρ
instance Record X where
  fold f_X _     = f_X
instance (Record ρ) ⇒ Record (ρ :& ν ::: ς) where
  fold f_X f_(:&) = f_(:&) $ fold f_X f_(:&)
```

Figure 7.12.: Definition of class *Record* with method *fold*

of the combinator for specific record schemes are constructed. The X-alternative is the specialization for the empty record scheme. The (:&)-alternative produces specializations for non-empty record schemes $\rho :\& \nu ::: \varsigma$ from the specializations for the respective schemes ρ.

It is clear from Figure 7.12 that the resulting combinator has a type $(Record\ \rho) \Rightarrow \theta\ \rho$, where θ cannot be an arbitrary type-level function, but has to be a type. Therefore, most record combinators cannot be implemented as *fold* applications. For example, to implement *multiProduce* as a result of *fold*, we would have to set θ to

$$\lambda\rho \rightarrow (\rho\ (TypeStyle\ Reg) \rightarrow IO\ (\rho\ (TypeStyle\ DSignal)))\ .$$

However, this type-level function is not a Haskell type.

It seems obvious to use emulation of type-level abstractions again to solve this problem. However, this does not work. In the type of *fold*, we would have to replace every type-level application of θ to some ρ by $App\ \theta\ \rho$. As a result, *fold*'s type would contain θ only as an index of the type synonym family *App*. Since type synonym families are not necessarily injective, this would mean that whenever *fold* is used, the concrete substitute for θ could not be deduced.

Our solution is to introduce wrapper types that are isomorphic to the type-level functions we actually want to use. For every inductively defined combinator χ of a type $(Record\ \rho) \Rightarrow \tau$, we introduce a type constructor Θ as follows:

```
newtype Θ ρ = Θ τ
```

Then we use *fold* to generate the wrapped combinator $\Theta\ \chi$. This is possible since that combinator has the type $(Record\ \rho) \Rightarrow \Theta\ \rho$, and Θ is a proper substitute for θ. Finally, we extract χ from $\Theta\ \chi$.

Figure 7.13 presents an implementation of *multiProduce* that is based on *fold*. Note that the declarations of the functions u_X and $u_{(:\&)}$ are very similar to the declarations of *multiProduce* in the two instance declarations of Figure 7.11. Alas, the wrapping and unwrapping of combinators makes the new implementation more verbose. This kind of overhead is our reason to not use wrapper types for record styles.

```
newtype Θ_multiProduce ρ = Θ_multiProduce
                             (ρ (TypeStyle Reg) → IO (ρ (TypeStyle DSignal)))
multiProduce :: (Record ρ) ⇒ ρ (TypeStyle Reg) → IO (ρ (TypeStyle DSignal))
multiProduce = let
                 Θ_multiProduce c = fold f_X f_(:&)
               in c where
  f_X :: Θ_multiProduce X
  f_X = Θ_multiProduce u_X
  u_X :: X (TypeStyle Reg) → IO (X (TypeStyle DSignal))
  u_X X = return X
  f_(:&) :: (Record ρ) ⇒ Θ_multiProduce ρ → Θ_multiProduce (ρ :& ν ::: ς)
  f_(:&) (Θ_multiProduce c) = Θ_multiProduce (u_(:&) c)
  u_(:&) :: (Record ρ) ⇒ (ρ (TypeStyle Reg) → IO (ρ (TypeStyle DSignal))) →
                         (ρ :& ν ::: ς) (TypeStyle Reg)                  →
                         IO ((ρ :& ν ::: ς) (TypeStyle DSignal))
  u_(:&) c (r̂ :& n := r) = do
                             x̂̂ ← c r̂
                             ẍ ← produce r
                             return (x̂̂ :& n := ẍ)
```

Figure 7.13.: Implementation of *multiProduce* based on *fold*

7.4.2. Is It Really a Fold?

It might not be immediately clear that the *fold* method of *Record* is actually a fold operator. Let us compare it with a fold over lists. We define a list data type that uses a "snoc" instead of a "cons" as the constructor for the non-empty case:

$$\textbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Snoc}\ (\mathit{List}\ \alpha)\ \alpha$$

The fold operator for *List* has the type

$$\theta \to (\theta \to \alpha \to \theta) \to \mathit{List}\ \alpha \to \theta\ .$$

It differs from the fold operator over record schemes in some important points:

1. The second argument of the list fold is a *Snoc* alternative. A *Snoc* alternative receives the last element of the respective list as its second argument. On the other hand, a (:&)-alternative does not get the name and sort of the last record field as arguments. Instead, the type of the (:&)-alternative uses universal quantification over all names and sorts.

2. The list fold has a third argument, which is the list to be folded. The *fold* method of *Record* does not receive the record scheme as an argument but uses universal quantification over all record schemes instead.

3. The θ-types of the record scheme fold have a record scheme parameter. Therefore, the type of a fold result may depend on the folded record scheme. In addition, the type of the (:&)-alternative needs a universal quantification over all record schemes. On the other hand, the θ-types of the list fold are not parameterized.

The first and the second difference are not fundamental, since universally quantified types are equivalent to dependent function types. Let α be a type variable, ξ be a kind, and τ' be a type that may have free occurrences of α. The universally quantified type $\forall\alpha :: \xi.\tau'$ is equivalent to the dependent function type $(\alpha :: \xi) \to \tau'$. This is the type of all functions that map each type τ of kind ξ to a corresponding value of type $\tau'[\tau/\alpha]$. Note the peculiarity that these functions take types as arguments, not values.

There is the additional fact that a type $\forall\alpha{::}\xi.\tau \to \tau'$ is isomorphic to $\tau \to \forall\alpha{::}\xi.\tau'$ if α does not occur free in τ. We can use this and the above-mentioned equivalence to transform the type of the record scheme fold into the equivalent dependent type

$$\theta\ X \to (\forall\rho{::}\Xi_{\mathit{Record}}.\theta\ \rho \to (\nu{::}*) \to (\varsigma{::}*) \to \theta\ (\rho{:}\&\nu{:::}\varsigma)) \to (\rho{::}\Xi_{\mathit{Record}}) \to \theta\ \rho\ .$$

Thus, the identifier Ξ_{Record} denotes a kind covering all *Record* instances. So a type $\forall\alpha :: \Xi_{\mathit{Record}}.\tau$ is equivalent to $\forall\alpha.(\mathit{Record}\ \alpha) \Rightarrow \tau$.

We can get rid of the third of the above differences by generalizing the fold operator over lists. For this generalization, we need a true dependent type system,

so that types can be parameterized by values, and we can universally quantify over values. The generalized fold has the type

$$\theta\ \mathit{Nil} \to (\forall \mathit{xs} :: \mathit{List}\ \alpha.\theta\ \mathit{xs} \to (x :: \alpha) \to \theta\ (\mathit{Snoc}\ \mathit{xs}\ x)) \to (\mathit{xs} :: \mathit{List}\ \alpha) \to \theta\ \mathit{xs}\ \ .$$

Note that this allows the fold result to have a type that depends on the folded list. The generalized list fold operator corresponds directly to the dependently-typed fold operator over record schemes.

7.5. Record Conversion

Our implementation of records imposes a total order on the fields of each record. While this is useful for distinguishing fields of the same name, it is undesirable otherwise. We want to be able to ignore such superfluous order. Furthermore, it is often beneficial if we can automatically ignore record fields we are not interested in. That way, record operations can be made more general. For example, a function application *multiProduce* $\hat{r}$ can also produce records that lack some of the fields that are specified by the registration action record $\hat{r}$. Therefore, we introduce a conversion operator for records that is able to reorder and drop fields.

7.5.1. Equivalence and Convertibility

Let us look at the special case of records that do not contain multiple fields of the same name. Such records denote mappings from names to values. We call these mappings the meanings of the respective records and write $[\![\hat{x}]\!]$ for the meaning of a record $\hat{x}$. We say that two records $\hat{x}_1$ and $\hat{x}_2$ are equivalent, written $\hat{x}_1 \approx \hat{x}_2$, if and only if they have the same meaning. So two records are equivalent if they only differ in the order of fields. A record $\hat{x}$ can be converted into a record $\hat{x}'$, written $\hat{x} \lesssim \hat{x}'$, if and only if

$$\mathrm{dom}[\![\hat{x}]\!] \supseteq \mathrm{dom}[\![\hat{x}']\!] \wedge \forall \nu \in \mathrm{dom}[\![\hat{x}']\!] : [\![\hat{x}]\!](\nu) = [\![\hat{x}']\!](\nu)\ \ .$$

So record conversion may reorder and drop fields arbitrarily. Note that $\hat{x}_1 \approx \hat{x}_2$ holds if and only if $\hat{x}_1 \lesssim \hat{x}_2 \wedge \hat{x}_2 \lesssim \hat{x}_1$.

Now, we want to also consider records that contain several fields of the same name. First, we modify the record semantics. The meaning of a record is now a function that maps each potential name, that is, each type of kind $*$, to the list of values that the record assigns to that name. The order of the values in the list matches the order of their respective fields in the record. Names that do not occur in the record are mapped to the empty list. Again, $\hat{x}_1 \approx \hat{x}_2$ shall hold if and only if $[\![\hat{x}_1]\!] = [\![\hat{x}_2]\!]$. So two records are equivalent if they contain the same fields and fields of the same name occur in the same order.

Having more values per name means that we cannot identify a field solely by its name anymore. We choose to identify a field of a record $\hat{x}$ by its name ν and the index of its value in $[\![\hat{x}]\!](\nu)$. Thereby, we index the values in $[\![\hat{x}]\!](\nu)$ backwards,

so that the first element gets the largest and the last element gets the smallest index. This will make the implementation of record conversion easier. An ordinary front-to-back indexing would not play well with the fact that (:&) appends fields instead of prepending them. We can now reformulate the criterion for record equivalence. Two records are equivalent if and only if they contain the same fields and the fields have the same indices in both records.

We want to ensure that after a record conversion, fields are identified in the same way as they were identified before. So a field must keep its index during conversion. We define record convertibility such that $\hat{x} \lesssim \hat{x}'$ holds if and only if for each ν of kind $*$, $[\![\hat{x}']\!](\nu)$ is a suffix of $[\![\hat{x}]\!](\nu)$. Again, we have the fact that $\hat{x}_1 \approx \hat{x}_2$ is equivalent to $\hat{x}_1 \lesssim \hat{x}_2 \wedge \hat{x}_2 \lesssim \hat{x}_1$.

We can see a record scheme as a kind of record itself by treating sorts as values and types of the form $\nu ::: \varsigma$ as fields with name ν and value ς. That way, we can extend $[\![\cdot]\!]$, $\approx$, and $\lesssim$ to schemes.

7.5.2. Implementation of Record Conversion

Let ρ be a record scheme, σ be a record style and $\hat{x}$ be a record of type $\rho\,\sigma$. There is a bijection between the sets $\{\rho' \mid \rho \lesssim \rho'\}$ and $\{\hat{x}' \mid \hat{x} \lesssim \hat{x}'\}$ such that for each concrete ρ' and corresponding $\hat{x}'$, $\hat{x}'$ has the type $\rho'\,\sigma$. The idea is that for each ρ', we can generate the corresponding $\hat{x}'$ from $\hat{x}$ by performing the same reorderings and droppings that we use to transform ρ into ρ'. So while a record $\hat{x}$ can usually be converted into different records $\hat{x}'$, we can select the desired conversion result via its scheme. We will use this in the implementation of record conversion.

We define a class *Convertible* with two parameters such that a pair of record schemes ρ and ρ' is an instance of *Convertible* if and only if $\rho \lesssim \rho'$. *Convertible* contains a method *convert* of type

$$(\mathit{Convertible}\ \rho\ \rho') \Rightarrow \rho\,\sigma \to \rho'\,\sigma\ .$$

This type implies that for each record $\hat{x}$ of type $\rho\ \sigma$, *convert* $\hat{x}$ has every type $\rho'\ \sigma$ for which $\rho \lesssim \rho'$ holds. For each concrete ρ', the type-restricted expression *convert* $\hat{x} :: \rho'\ \sigma$ yields the conversion result $\hat{x}'$ that corresponds to ρ' according to the above-mentioned bijection.

Figure 7.14 shows the definition of *Convertible*. This definition uses induction on the scheme of the conversion result. It employs a helper class *Separation*. A quadruple of two record schemes ρ and ρ_s, a name ν, and a sort ς is an instance of *Separation* if and only if the last $(:::)$-type in ρ that has name ν is $\nu ::: \varsigma$, and removing this type from ρ yields ρ_s. The *separate* method extracts the last field of name ν from the given record and yields this field together with the remaining record.

The instance declarations of *Separation* overlap, so that we need support for overlapping instances from the compiler. The HList library avoids overlapping when defining record conversion. In HList, field names are not represented by arbitrary types, but essentially by type-level naturals. HList defines an equality

$$
\begin{aligned}
&\textbf{class } \mathit{Convertible}\ \rho\ \rho' \textbf{ where}\\
&\quad \mathit{convert} :: \rho\ \sigma \to \rho'\ \sigma\\
&\textbf{instance } \mathit{Convertible}\ \rho\ X \textbf{ where}\\
&\quad \mathit{convert}\ \_ = X\\
&\textbf{instance } (\mathit{Separation}\ \rho\ \rho_\mathrm{s}\ \nu\ \varsigma, \mathit{Convertible}\ \rho_\mathrm{s}\ \rho'_\mathrm{s}) \Rightarrow\\
&\qquad \mathit{Convertible}\ \rho\ (\rho'_\mathrm{s} :\& \nu ::: \varsigma) \textbf{ where}\\
&\quad \mathit{convert}\ \hat{x} = \textbf{let}\\
&\qquad\qquad (\hat{x}_\mathrm{s}, f) = \mathit{separate}\ \hat{x}\\
&\qquad\quad \textbf{in } \mathit{convert}\ \hat{x}_\mathrm{s} :\& f\\
&\textbf{class } \mathit{Separation}\ \rho\ \rho_\mathrm{s}\ \nu\ \varsigma \mid \rho\ \nu \to \rho_\mathrm{s} \textbf{ where}\\
&\quad \mathit{separate} :: \rho\ \sigma \to (\rho_\mathrm{s}\ \sigma, (\nu ::: \varsigma)\ \sigma)\\
&\textbf{instance } (\varsigma \sim \varsigma') \Rightarrow\\
&\qquad \mathit{Separation}\ (\rho :\& \nu ::: \varsigma)\ \rho\ \nu\ \varsigma' \textbf{ where}\\
&\quad \mathit{separate}\ (\hat{x} :\& f) = (\hat{x}, f)\\
&\textbf{instance } (\mathit{Separation}\ \rho\ \rho_\mathrm{s}\ \nu'\ \varsigma', (\rho_\mathrm{s} :\& \nu ::: \varsigma) \sim \rho') \Rightarrow\\
&\qquad \mathit{Separation}\ (\rho :\& \nu ::: \varsigma)\ \rho'\ \nu'\ \varsigma' \textbf{ where}\\
&\quad \mathit{separate}\ (\hat{x} :\& f) = \textbf{let}\\
&\qquad\qquad\qquad (\hat{x}_\mathrm{s}, f') = \mathit{separate}\ \hat{x}\\
&\qquad\qquad\quad \textbf{in } (\hat{x}_\mathrm{s} :\& f, f')
\end{aligned}
$$

Figure 7.14.: Implementation of record conversion

check for type-level naturals that turns each pair of naturals into a corresponding type-level boolean. Such a boolean can be used to select the appropriate instance.

The downside of this approach is that name declarations become more complicated. A solution is to use a general type equality check instead of a check that only works for natural numbers. The HList authors implemented such a check, but they needed overlapping instances to do so. Since we would use such an equality check only in the instance declarations of *Separation*, we decided to use overlapping instances directly in the implementation of record conversion.

Separation uses a functional dependency to specify that the scheme of the source record and the name of the extracted field uniquely determine the scheme of the remaining record. It seems more sensible to use a type synonym family to specify this dependency. After all, we already used the feature of type synonym families to emulate type-level abstractions in Subsection 7.3.2. The problem is that the instance declarations of *Separation* overlap, which would result in overlapping instance declarations for the type synonym family that we would introduce. However, overlapping is forbidden for type synonym families.

Now, let us look at the type equality constraint $\varsigma \sim \varsigma'$ in the first instance declaration of *Separation*. This equality constraint ensures that the actual sort of the extracted field equals the specification of the extracted field's sort. Normally, we could eliminate this equality constraint by using a single type variable instead of the two different variables ς and ς'. That is, we could replace

$$(\varsigma \sim \varsigma') \Rightarrow \mathit{Separation}\ (\rho :\& \nu ::: \varsigma)\ \rho\ \nu\ \varsigma'$$

by

$$\mathit{Separation}\ (\rho :\& \nu ::: \varsigma)\ \rho\ \nu\ \varsigma\ .$$

However, because of our use of overlapping instances, this transformation would change the meaning of the program.

The original instance declaration head matches whenever the last name of the record scheme equals the name of the extracted field. If the last sort is not the one that is specified as the sort of the extracted field, the equality constraint is not fulfilled, and we get a type error. This is in line with our specification of *Separate* above.

If we would eliminate the equality constraint, the instance declaration head would not match in case the last name equals the name of the extracted field, but the last sort is different from the required sort. However, the head of the second instance declaration would match in this case. Therefore, the *separate* method would not extract the last field that has the respective name but the last field whose name and sort are the required ones. This would be contrary to our specification of *Separation* and would lead to a bogus implementation of *Convertible*.

An advantage of the solution with the type equality constraint is that we only need a name to identify the field that we want to extract, not a sort. So the sort of the extracted field can be unknown initially and then determined by the equality

constraint. This is useful, for example, in record pattern matching, which we will describe in the next subsection.

Support for type equality constraints was introduced into GHC as part of the type family extension since type equality constraints are often helpful when working with type synonym families. Our usage of equality constraints shows that they are also useful without type families. We therefore argue that they should be treated as a separate extension to the core language, independent of type families.

7.5.3. Record Pattern Matching

Since records are values of algebraic data types, we can use pattern matching to access the values of their fields. However, a pattern must be of the same type as the record that is matched against it. So the pattern must contain one subpattern for each record field, and these subpatterns must occur in the order their corresponding fields occur in. This is a major drawback in comparison to other record systems. We want to be able to reorder and drop fields automatically during pattern matching.

If we replace a record $\hat{x}$ with the record *convert* $\hat{x}$, the type of the record is changed from $\rho\ \sigma$ to $(\mathit{Convertible}\ \rho\ \rho') \Rightarrow \rho'\ \sigma$. We can specify the concrete scheme of the conversion result by matching *convert* $\hat{x}$ against a pattern of the form

$$X :\& \nu_1 := \pi_1 :\& \ldots :\& \nu_n := \pi_n \ ,$$

where ν_1 through ν_n are concrete names, and π_1 through π_n are arbitrary patterns. We do not need to assign sorts to the patterns π_i. The reason is that instance selection for *Convertible* and *Separate* only depends on names, and the π_i automatically get the correct sorts from ρ because of the equality constraint $\varsigma \sim \varsigma'$.

Support for record pattern matching is an advantage over the HList library. Name types in HList contain no values apart from $\bot$. So there are no patterns that match individual names. As a consequence, HList cannot provide pattern matching for records.

8. First Class Subkinds

The *fold* operator from Subsection 7.4.1 can only generate combinators that work on all record schemes. However, there are record combinators that only work on record schemes whose sorts fulfill certain conditions. To solve this problem, we develop a technique for emulating subkinds, including subkind polymorphism. This technique is implemented in the *kinds* package[1]. The contents of this chapter are largely taken from our publication on records [12]. See Appendix A regarding the permission to reproduce parts of this work here.

Let us look at an example again. In Subsection 5.3.3, we introduced the type *SuffixFun* of suffix functions and its data constructors *OSF* and *SSF* for constructing nullary and non-nullary suffix functions, respectively. We want to implement a combinator *multiOSF* that turns a record of suffixes into a record of nullary suffix functions. We require that all suffixes in the argument record share the same start time. Therefore, the sort of a single field must not specify a start time, but only the respective suffix type constructor and its second argument. Since a sort must be a single type instead of two types, we apply the *Of* type constructor from Subsection 5.3.3 to the suffix type constructor and its second argument to create one type out of these two ingredients.

The style of the suffix record that *multiOSF* receives turns each type $\sigma\ `Of`\ \alpha$ into $\sigma\ \tau\ \alpha$ where τ is the start time parameter for the whole record. We implement this style as follows:

```
data SuffixStyle τ
type instance App (SuffixStyle τ) (σ `Of` α) = σ τ α
```

Now, it seems obvious to give *multiOSF* the type

$$(Record\ \rho) \Rightarrow \rho\ (SuffixStyle\ \tau) \rightarrow \rho\ (TypeStyle\ (SuffixFun\ \tau))\ .$$

However, *multiOSF* can only work with record schemes whose sorts are of the form $\sigma\ `Of`\ \alpha$ with σ being an instance of *Suffix*. So the above type for *multiOSF* is too general, since it allows arbitrary record schemes.

It is also not possible to implement *multiOSF* via the *fold* operator defined in Subsection 7.4.1. This *fold* operator requires a (:&)-alternative that places no restrictions on the last sort of the record scheme. Say $\Theta_{multiOSF}$ is the θ-type of the inductive definition of *multiOSF*. Then the type of *fold*'s second argument is

$$\forall \rho\ \nu\ \varsigma.(Record\ \rho) \Rightarrow \Theta_{multiOSF}\ \rho \rightarrow \Theta_{multiOSF}\ (\rho :\& \nu ::: \varsigma)\ .$$

[1] See `http://hackage.haskell.org/package/kinds`.

```
data Kind_SuffixShape
data Kind_FunShape
data Kind_Star
instance (Suffix σ) ⇒ Inhabitant Kind_SuffixShape (σ `Of` α)
instance (Suffix σ) ⇒ Inhabitant Kind_FunShape   (σ `Of` α)
instance (Suffix σ) ⇒ Inhabitant Kind_FunShape   (σ `Of` α ↦ φ)
instance              Inhabitant Kind_Star       α
```

Figure 8.1.: Emulation of example subkinds

However, the (:&)-alternative in the definition of *multiOSF* has to apply *OSF* to the value of the last field, so that it has the less general type

$$\forall \rho\ \nu\ \sigma\ \alpha.(\mathit{Record}\ \rho, \mathit{Suffix}\ \sigma) \Rightarrow \Theta_{\mathit{multiOSF}}\ \rho \rightarrow \Theta_{\mathit{multiOSF}}\ (\rho :\&\ \nu ::: \sigma\ `\mathit{Of}`\ \alpha)\ .$$

To solve this problem, we introduce the notion of subkind. Subkinds are the kind-level analogon to subtypes. So a subkind of a kind ξ denotes a set of types that all have kind ξ. We refine the *Record* class such that it supports inductive definitions over all record schemes whose sorts have a given subkind of kind $*$. In the case of *multiOSF*, we use the subkind of all suffix shapes. For functions that are defined on all record schemes, we use $*$ itself.

8.1. Emulation of Subkinds

Haskell has no built-in support for subkinds. However, we can use existing type system features to emulate subkinds of a fixed base kind. In this thesis, we only deal with subkinds of kind $*$. Subsection 8.1.1 presents a simple technique for subkind emulation, and Subsection 8.1.2 shows how to construct a kind-aware *fold*. Subsection 8.1.3 points out problems with the presented technique, which are solved in Section 8.2.

8.1.1. A Simple Emulation Technique

We represent subkinds by types. That way, subkinds are first class citizens at the type level. In addition, we can use type polymorphism to emulate subkind polymorphism. We introduce a two-parameter class *Inhabitant* with no methods. Each pair of a subkind representation and a type that has the respective subkind is an instance of *Inhabitant*. Figure 8.1 shows the emulation of three example subkinds. *SuffixShape* is the kind of all suffix shapes, *FunShape* is the kind of all φ for which *SuffixFun* φ contains values apart from $\bot$, and *Star* denotes kind $*$, which is, of course, a subkind of itself.

$$
\begin{array}{lll}
\textbf{subkind } \mathit{SuffixShape} & = & (\mathit{Signal}\ \sigma) \Rightarrow \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \\
\textbf{subkind } \mathit{FunShape} & = & (\mathit{Signal}\ \sigma) \Rightarrow \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \\
 & \mid & (\mathit{Signal}\ \sigma) \Rightarrow \sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \varphi \\
\textbf{subkind } \mathit{Star} & = & \alpha
\end{array}
$$

Figure 8.2.: Definition of example subkinds

Now that we have looked at some examples, let us discuss the general picture. Imagine we had a new language construct for declaring subkinds of kind $*$. The declaration

$$\textbf{subkind } \Xi = \Gamma_1 \Rightarrow \tau_1 \mid \ldots \mid \Gamma_n \Rightarrow \tau_n$$

introduces a subkind Ξ. Thereby, Γ_1 through Γ_n are contexts, and τ_1 through τ_n are types. They have to satisfy the following conditions:

- $\mathrm{FV}(\Gamma_i) \subseteq \mathrm{FV}(\tau_i)$ for all i with $1 \leq i \leq n$.
- For all i and j with $1 \leq i < j \leq n$, the types τ_i and τ_j cannot be unified.

The declared subkind Ξ covers all types τ for which there is an i and a variable assignment σ such that $\tau = \sigma(\tau_i)$ and the context $\sigma(\Gamma_i)$ holds.

To emulate the above subkind declaration, we first introduce an empty data type Kind_Ξ:

$$\textbf{data } \mathit{Kind}_\Xi$$

Afterwards, we provide an instance declaration of the following form for every i with $1 \leq i \leq n$:

$$\textbf{instance } \Gamma_i \Rightarrow \mathit{Inhabitant}\ \mathit{Kind}_\Xi\ \tau_i$$

Of course, we can use subkind declarations to introduce the three example subkinds. Such subkind declarations are shown in Figure 8.2. If we transform them into data type and instance declarations, we end up with the code from Figure 8.1.

8.1.2. Records with Kinded Sorts

We change the *Record* class such that we can specify a subkind that all sorts have to belong to. We add a parameter to *Record* such that the class assertion $\mathit{Record}\ \varkappa\ \rho$ means that ρ is a record scheme that contains only sorts of the subkind represented by $\varkappa$. The new definition of *Record* is shown in Figure 8.3. It differs from the original one in the following points:

- All references to the *Record* class contain an additional parameter $\varkappa$.

$$
\begin{array}{l}
\textbf{class } \mathit{Record}\ \varkappa\ \rho\ \textbf{where} \\
\quad \mathit{fold} :: \theta\ X \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \to \\
\qquad\quad (\forall \rho\ \nu\ \varsigma.(\mathit{Record}\ \varkappa\ \rho, \mathit{Inhabitant}\ \varkappa\ \varsigma) \Rightarrow \theta\ \rho \to \theta\ (\rho :\& \nu ::: \varsigma)) \to \\
\qquad\quad \theta\ \rho \\
\textbf{instance } \mathit{Record}\ \varkappa\ X\ \textbf{where} \\
\quad \mathit{fold}\ f_X\ \_ \quad = f_X \\
\textbf{instance } (\mathit{Record}\ \varkappa\ \rho, \mathit{Inhabitant}\ \varkappa\ \varsigma) \Rightarrow \mathit{Record}\ \varkappa\ (\rho :\& \nu ::: \varsigma)\ \textbf{where} \\
\quad \mathit{fold}\ f_X\ f_{(:\&)} = f_{(:\&)}\ \$\ \mathit{fold}\ f_X\ f_{(:\&)}
\end{array}
$$

Figure 8.3.: Definition of class *Record* with kinded sorts

- The head of the second instance declaration contains an additional assertion $\mathit{Inhabitant}\ \varkappa\ \varsigma$, which enforces that sorts are of the specified subkind.
- The type of the second argument of *fold* contains a further class assertion $\mathit{Inhabitant}\ \varkappa\ \varsigma$, so that (:&)-alternatives only have to work with schemes whose last sort has the respective subkind.

Having the new *Record* definition, we give *multiOSF* the type

$$(\mathit{Record}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \rho) \Rightarrow \rho\ (\mathit{SuffixStyle}\ \tau) \to \rho\ (\mathit{TypeStyle}\ (\mathit{SuffixFun}\ \tau)) \ .$$

8.1.3. Problems with the Current Approach

A problem with the new definition of *Record* is that the class parameter $\varkappa$ does not occur in the type of *fold* except in class assertions. So when *fold* is used in some expression, the actual $\varkappa$-parameter cannot be determined. This occurs, for example, in the declaration of *fold* for the (:&)-case. GHC complains that it cannot deduce the context $(\mathit{Record}\ \varkappa_1\ \rho)$ from the context

$$(\mathit{Record}\ \varkappa_2\ (\rho :\& \nu ::: \varsigma), \mathit{Record}\ \varkappa_2\ \rho, \mathit{Inhabitant}\ \varkappa_2\ \varsigma) \ .$$

The variable $\varkappa_1$ denotes the $\varkappa$-parameter of the *fold* in the expression $f_{(:\&)}$ \$ $\mathit{fold}\ f_X\ f_{(:\&)}$, while $\varkappa_2$ denotes the $\varkappa$-parameter of the *fold* in the left-hand side. GHC cannot see why both parameters should be equal.

There is a second, more serious, problem. Since Haskell classes are open, we cannot prevent subkinds from being extended. For example, someone could import our definition of the *SuffixShape* subkind and add the type *Bool* to this subkind using the following instance declaration:

$$\textbf{instance } \mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \mathit{Bool}$$

So we have no guarantee that $(\mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \varsigma)$ holds only for those ς that are of the form $\sigma\ `\mathit{Of}`\ \varsigma$ with $(\mathit{Suffix}\ \sigma)$.

This makes it impossible to define *multiOSF* using *fold*. The (:&)-alternative in the inductive definition of *multiOSF* has the most general type

$$\forall \rho\ \nu\ \sigma\ \alpha.(\mathit{Record}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \rho, \mathit{Suffix}\ \sigma) \Rightarrow \\ \Theta_{\mathit{multiOSF}}\ \rho \rightarrow \Theta_{\mathit{multiOSF}}\ (\rho :\& \nu ::: \sigma\ `\mathit{Of}`\ \alpha)\ .$$

This type is less general than the required type

$$\forall \rho\ \nu\ \varsigma.(\mathit{Record}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \rho, \mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \varsigma) \Rightarrow \\ \Theta_{\mathit{multiOSF}}\ \rho \rightarrow \Theta_{\mathit{multiOSF}}\ (\rho :\& \nu ::: \varsigma)\ .$$

In the next section, we present a technique for closing subkinds, which resolves this problem. As a side effect, we also get rid of the problem that the $\varkappa$-parameter of a *fold* application cannot be inferred.

8.2. Closing Subkinds

To close a subkind means to ensure that no further inhabitants can be added to it. So far, we relied solely on the type class *Inhabitant* for specifying the inhabitants of a kind. The problem is that type classes are open. In this section, we show how later extensions of subkinds can prevented nevertheless.

8.2.1. Isomorphisms to the Rescue

Let us look at some examples again. To close the *SuffixShape* subkind, we must ensure that the set of types ς with $(\mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \varsigma)$ is the same as the set of types $\sigma\ `\mathit{Of}`\ \alpha$ with $(\mathit{Suffix}\ \sigma)$. We can enforce this by making sure that universal quantification over all ς with $(\mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \varsigma)$ is the same as universal quantification over all suffix shapes. That is, for any type-level function F, the types

$$\forall \varsigma.(\mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \varsigma) \Rightarrow F\ \varsigma$$

and

$$\forall \sigma\ \alpha.(\mathit{Suffix}\ \sigma) \Rightarrow F\ (\sigma\ `\mathit{Of}`\ \alpha)$$

are isomorphic. If we set F to

$$\lambda\varsigma \rightarrow \forall \rho\ \nu.(\mathit{Record}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \rho) \Rightarrow \Theta_{\mathit{multiOSF}}\ \rho \rightarrow \Theta_{\mathit{multiOSF}}\ (\rho :\& \nu ::: \varsigma)\ ,$$

those types correspond to the required and inferred type of the (:&)-alternative of the *multiOSF* definition. If these are isomorphic, the former cannot be more general than the latter anymore. This allows us to define *multiOSF* using *fold*.

To close the *FunShape* subkind, we have to make sure that universal quantification over all ς with $(\mathit{Inhabitant}\ \mathit{Kind}_{\mathit{FunShape}}\ \varsigma)$ is the same as universal quantification over all types of subkind *FunShape*. The question is how the latter can be expressed. After all, the types of *FunShape* do not all share a common structure. On the one hand, we have the nullary function shapes, which have the form $\sigma\ `\mathit{Of}`\ \alpha$ with

(*Suffix* σ), on the other hand, we have the non-nullary shapes, which have the form σ ‘*Of*‘ $\alpha \mapsto \varphi$ with (*Suffix* σ). However, we can universally quantify over only the nullary shapes and also over only the non-nullary shapes. We will now show how we can use this to universally quantify over all types of subkind *FunShape*.

As mentioned in Subsection 7.4.2, a type $\forall\alpha :: \xi.\tau'$ is isomorphic to the dependent function type $(\alpha :: \xi) \to \tau'$. Say we split ξ into two non-overlapping subkinds ξ_1 and ξ_2. Then we can split each function of type $(\alpha :: \xi) \to \tau'$ into two functions of types $(\alpha :: \xi_1) \to \tau'$ and $(\alpha :: \xi_2) \to \tau'$, respectively. In addition, we can merge two such functions to get back the corresponding function of type $(\alpha :: \xi) \to \tau'$. So a type $\forall\alpha :: \xi.\tau'$ is isomorphic to the type

$$(\forall\alpha :: \xi_1.\tau', \forall\alpha :: \xi_2.\tau') \ .$$

Therefore, we can close the *FunShape* subkind by ensuring that there is an isomorphism between

$$\forall\varsigma.(\mathit{Inhabitant\ Kind}_{\mathit{FunShape}}\ \varsigma) \Rightarrow F\ \varsigma$$

and

$$(\forall\sigma\ \alpha.(\mathit{Suffix}\ \sigma) \Rightarrow F\ (\sigma\ \text{‘}\mathit{Of}\text{‘}\ \alpha), \forall\sigma\ \alpha\ \varphi.(\mathit{Suffix}\ \sigma) \Rightarrow F\ (\sigma\ \text{‘}\mathit{Of}\text{‘}\ \alpha \mapsto \varphi))$$

for every type-level function F.

It is now easy to see how we can close subkinds in general. Say Ξ is a subkind that is declared as follows:

$$\textbf{subkind}\ \Xi = \Gamma_1 \Rightarrow \tau_1 \mid \ldots \mid \Gamma_n \Rightarrow \tau_n$$

Then we have to make sure that for all type-level functions F,

$$\forall\varsigma.(\mathit{Inhabitant\ Kind}_{\Xi}\ \varsigma) \Rightarrow F\ \varsigma$$

is isomorphic to

$$(\forall A_1.\Gamma_1 \Rightarrow F\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow F\ \tau_n)$$

where for each i with $1 \leq i \leq n$, A_i is a whitespace-separated sequence of the free variables of τ_i.

8.2.2. Ensuring the Existence of the Isomorphisms

Let us see how we can ensure that the abovementioned isomorphisms exist. It is sufficient to enforce the existence of isomorphisms only for those type-level functions that can be represented without using type-level abstractions, that is, for all types of kind $* \to *$. The reason is that for any type-level function F, we can introduce a type Φ that is isomorphic to F as follows:

$$\textbf{newtype}\ \Phi\ \alpha = \Phi\ (F\ \alpha)$$

If f is an isomorphism for Φ, the function $\Phi^{-1} \circ f \circ \Phi$ is an isomorphism for F.

We do not allow different isomorphisms for different types. Instead, we require a single isomorphism for all types ϕ of kind $* \to *$. We do so by demanding the existence of two functions

$$\overrightarrow{f_\Xi} :: (\forall \varsigma.(\mathit{Inhabitant}\ \mathit{Kind}_\Xi\ \varsigma) \Rightarrow \phi\ \varsigma) \to (\forall A_1.\Gamma_1 \Rightarrow \phi\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow \phi\ \tau_n)$$

and

$$\overleftarrow{f_\Xi} :: (\forall A_1.\Gamma_1 \Rightarrow \phi\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow \phi\ \tau_n) \to (\forall \varsigma.(\mathit{Inhabitant}\ \mathit{Kind}_\Xi\ \varsigma) \Rightarrow \phi\ \varsigma)$$

with $\overrightarrow{f_\Xi} \circ \overleftarrow{f_\Xi} = \overleftarrow{f_\Xi} \circ \overrightarrow{f_\Xi} = \mathit{id}$.

We will now show how we can actually enforce the existence of such functions $\overrightarrow{f_\Xi}$ and $\overleftarrow{f_\Xi}$. Let us first look at the functions $\overrightarrow{f_\Xi}$, which perform "forward conversions". We introduce a type class *Kind* of all subkind representations. *Kind* contains a method *closed* whose implementations perform the forward conversions of the respective subkinds. Furthermore, we change the definition of *Record* such that the $\varkappa$-parameter must be an instance of *Kind*. This ensures that if we use records with sorts of a certain subkind, there is a forward conversion for that subkind.

Say $\varkappa$ is the type variable used in the head of the class declaration of *Kind*. Then the argument type of *closed* is

$$\forall \varsigma.(\mathit{Inhabitant}\ \varkappa\ \varsigma) \Rightarrow \mathit{item}\ \varsigma\ .$$

The structure of the result type depends on the concrete subkind. So we cannot come up with a single result type that uses $\varkappa$ only as an ordinary type parameter. Instead, we have to use $\varkappa$ as a type index for selecting the particular result type. We introduce an associated data family [3] *All* for this purpose. For every subkind Ξ with alternatives $\Gamma_1 \Rightarrow \tau_1$ through $\Gamma_n \Rightarrow \tau_n$, the type $\mathit{All}\ \mathit{Kind}_\Xi$ is isomorphic to

$$\lambda\phi \to (\forall A_1.\Gamma_1 \Rightarrow \phi\ \tau_1, \ldots, \forall A_n.\Gamma_n \Rightarrow \phi\ \tau_n)$$

where the A_i are defined as above.

The complete class declaration of *Kind* is shown in Figure 8.4. For each subkind Ξ with alternatives $\Gamma_1 \Rightarrow \tau_1$ through $\Gamma_n \Rightarrow \tau_n$, we make Kind_Ξ an instance of *Kind* using an instance declaration of the following form:

instance *Kind* Kind_Ξ **where**
  data $\mathit{All}\ \mathit{Kind}_\Xi\ \phi = \mathit{All}_\Xi\ (\forall A_1.\Gamma_1 \Rightarrow \phi\ \tau_1)$
  $\ldots$
  $(\forall A_n.\Gamma_n \Rightarrow \phi\ \tau_n)$
  $\mathit{closed}\ x = \mathit{All}_\Xi\ x\ \ldots\ x$

The type of the argument of *closed* is

$$\forall \varsigma.(\mathit{Inhabitant}\ \mathit{Kind}_\Xi\ \varsigma) \Rightarrow \phi\ \varsigma\ .$$

$$\begin{aligned}&\textbf{class } \mathit{Kind}\ \varkappa \textbf{ where}\\&\quad \textbf{data } \mathit{All}\ \varkappa :: (* \to *) \to *\\&\quad \mathit{closed} :: (\forall \varsigma.(\mathit{Inhabitant}\ \varkappa\ \varsigma) \Rightarrow \phi\ \varsigma) \to \mathit{All}\ \varkappa\ \phi\end{aligned}$$

Figure 8.4.: Declaration of class *Kind*

$$\begin{aligned}&\textbf{instance } \mathit{Kind}\ \mathit{Kind}_{\mathit{SuffixShape}} \textbf{ where}\\&\quad \textbf{data } \mathit{All}\ \mathit{Kind}_{\mathit{SuffixShape}}\ \phi = \mathit{All}_{\mathit{SuffixShape}}\ (\forall \sigma\ \alpha.(\mathit{Suffix}\ \sigma) \Rightarrow \phi\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha))\\&\quad \mathit{closed}\ x = \mathit{All}_{\mathit{SuffixShape}}\ x\\&\textbf{instance } \mathit{Kind}\ \mathit{Kind}_{\mathit{FunShape}} \textbf{ where}\\&\quad \textbf{data } \mathit{All}\ \mathit{Kind}_{\mathit{FunShape}}\ \phi = \mathit{All}_{\mathit{FunShape}}\ (\forall \sigma\ \alpha.\quad (\mathit{Suffix}\ \sigma) \Rightarrow \phi\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha))\\&\qquad\qquad\qquad\qquad\qquad\qquad\quad (\forall \sigma\ \alpha\ \varphi.(\mathit{Suffix}\ \sigma) \Rightarrow \phi\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \varphi))\\&\quad \mathit{closed}\ x = \mathit{All}_{\mathit{FunShape}}\ x\ x\\&\textbf{instance } \mathit{Kind}\ \mathit{Kind}_{\mathit{Star}} \textbf{ where}\\&\quad \textbf{data } \mathit{All}\ \mathit{Kind}_{\mathit{Star}}\ \phi = \mathit{All}_{\mathit{Star}}\ (\forall \alpha.\phi\ \alpha)\\&\quad \mathit{closed}\ x = \mathit{All}_{\mathit{Star}}\ x\end{aligned}$$

Figure 8.5.: *Kind* instance declarations for example subkinds

On the right-hand side of the definition of *closed*, this type is specialized to the types $\forall A_i.\Gamma_i \Rightarrow \phi\ \tau_i$. These specializations are possible, because we have introduced an instance declaration of the following form for each i:

$$\textbf{instance } \Gamma_i \Rightarrow \mathit{Inhabitant}\ \mathit{Kind}_{\Xi}\ \tau_i$$

The instance declarations of *Kind* for *SuffixShape*, *FunShape*, and *Star* are shown in Figure 8.5.

Now, we will introduce a function that performs "backwards conversions". The type of this function is

$$(\mathit{Kind}\ \varkappa) \Rightarrow \mathit{All}\ \varkappa\ \phi \to (\forall \varsigma.(\mathit{Inhabitant}\ \varkappa\ \varsigma) \Rightarrow \phi\ \varsigma)\ .$$

A type $\tau \to (\forall \alpha.\Gamma \Rightarrow \tau')$ is equivalent to $\forall \alpha.\Gamma \Rightarrow \tau \to \tau'$ as long as α does not occur free in τ. So the backwards conversion function has also the type

$$(\mathit{Kind}\ \varkappa, \mathit{Inhabitant}\ \varkappa\ \varsigma) \Rightarrow \mathit{All}\ \varkappa\ \phi \to \phi\ \varsigma\ .$$

We add a context $(\mathit{Kind}\ \varkappa)$ to the class declaration of *Inhabitant* and declare the function for backwards conversion as a method of *Inhabitant*. The resulting code is shown in Figure 8.6. For a concrete subkind inhabitant ς, *specialize* converts from types with universal quantification over all inhabitants to the corresponding types that fix the inhabitant to ς. That is where *specialize* got its name from.

$$\textbf{class}\ (\mathit{Kind}\ \varkappa) \Rightarrow \mathit{Inhabitant}\ \varkappa\ \varsigma\ \textbf{where}$$
$$\quad \mathit{specialize} :: \mathit{All}\ \varkappa\ \phi \to \phi\ \varsigma$$

Figure 8.6.: Declaration of class *Inhabitant*

$$\textbf{instance}\ (\mathit{Suffix}\ \sigma) \Rightarrow \mathit{Inhabitant}\ \mathit{Kind}_{\mathit{SuffixShape}}\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha)\ \textbf{where}$$
$$\quad \mathit{specialize}\ (\mathit{All}_{\mathit{SuffixShape}}\ x) = x$$
$$\textbf{instance}\ (\mathit{Suffix}\ \sigma) \Rightarrow \mathit{Inhabitant}\ \mathit{Kind}_{\mathit{FunShape}}\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha)\ \textbf{where}$$
$$\quad \mathit{specialize}\ (\mathit{All}_{\mathit{FunShape}}\ x\ \_) = x$$
$$\textbf{instance}\ (\mathit{Suffix}\ \sigma) \Rightarrow \mathit{Inhabitant}\ \mathit{Kind}_{\mathit{FunShape}}\ (\sigma\ \text{`}\mathit{Of}\text{`}\ \alpha \mapsto \varphi)\ \textbf{where}$$
$$\quad \mathit{specialize}\ (\mathit{All}_{\mathit{FunShape}}\ \_\ x) = x$$
$$\textbf{instance}\ \mathit{Inhabitant}\ \mathit{Kind}_{\mathit{Star}}\ \alpha\ \textbf{where}$$
$$\quad \mathit{specialize}\ (\mathit{All}_{\mathit{Star}}\ x) = x$$

Figure 8.7.: *Inhabitant* instance declarations for example subkinds

For each subkind Ξ with alternatives $\Gamma_1 \Rightarrow \tau_1$ through $\Gamma_n \Rightarrow \tau_n$ and each i with $1 \leq i \leq n$, we need an instance declaration of the following form:

$$\textbf{instance}\ \Gamma_i \Rightarrow \mathit{Inhabitant}\ \mathit{Kind}_{\Xi}\ \tau_i\ \textbf{where}$$
$$\quad \mathit{specialize}\ (\mathit{All}_{\Xi}\ \_^{i-1}\ x\ \_^{n-i}) = x$$

Hereby, $\_^k$ stands for a whitespace-separated sequence of k wildcard patterns (_). Figure 8.7 shows the concrete instance declarations for our three example subkinds.

Of course, the class declarations of *Kind* and *Inhabitant* do not ensure that instance declarations are formed according to the rules described above. So there is no guarantee that $\mathit{closed} \circ \mathit{specialize} = \mathit{specialize} \circ \mathit{closed} = \mathit{id}$ holds in fact. However, this is a general problem with Haskell's class system. For example, sensible instance declarations of *Ord* have to fulfill the condition $(<) = \mathit{flip}\ (>)$, but the compiler cannot check whether they actually do.

8.2.3. Adapting the *Record* Class

Figure 8.8 shows the final definition of the *Record* class. This definition forces $\varkappa$-parameters to be instances of the *Kind* class. In addition, the type of *fold*'s second argument now uses the *All* data family. Thus, $\varkappa$ occurs in *fold*'s type not only in a context but also as a data family parameter. Therefore, actual $\varkappa$-parameters can now be inferred. Note that this would not be possible if *All* would be a type synonym family since type synonym families are not guaranteed to be injective.

```
newtype Expander θ ρ ν ς = Expander (θ ρ → θ (ρ :& ν ::: ς))
class (Kind ϰ) ⇒ Record ϰ ρ where
  fold :: θ X → (∀ρ ν.(Record ϰ ρ) ⇒ All ϰ (Expander θ ρ ν)) → θ ρ
instance (Kind ϰ) ⇒ Record ϰ X where
  fold f_X _ = f_X
instance (Record ϰ ρ, Inhabitant ϰ ς) ⇒ Record ϰ (ρ :& ν ::: ς) where
  fold f_X e = let
                 Expander f_(:&) = specialize e
               in f_(:&) $ fold f_X e
```

Figure 8.8.: Final definition of class *Record*

The use of *All* makes the definition of a wrapper type *Expander* necessary. For all θ, ρ, and ν, the type *Expander* θ ρ ν is isomorphic to the type-level function

$$\lambda\varsigma \rightarrow (\theta\ \rho \rightarrow \theta\ (\rho :\& \nu ::: \varsigma))\ .$$

9. Conclusions and Further Work

In this thesis, we discussed several topics related to Functional Reactive Programming:

- We surveyed FRP implementations from other authors based on a common interface and accompanying semantics.
- We presented two own contributions to the field of FRP, namely, the realization of start time consistency through the type system and the vista data structure for implementing discrete suffixes. Together, they lead to a system for discrete FRP that has a simple, yet efficient and semantically correct implementation.
- We developed a generic record system that features record type families, a fold operator over record schemes, and a generic record conversion operator. The record system allows one to define a wide variety of generic record combinators that are statically typed. We emulated polymorphic subkinds to give the record system additional strength.

In the remainder of this chapter, we discuss ideas for further work. Parts of this discussion are taken from earlier works of the author [13, 12]. See Appendix A for important information on copyright.

9.1. Further Work on FRP

We have not discussed how continuous suffixes can be implemented in a push-based setting. Actually, the push-based approach does not play well with continuity, since it relies on interesting things only happening at discrete times. A solution, developed independently by Elliott [7] and us, is to combine a push-based implementation of segmented suffixes with a pull-based implementation of time-varying values, leading to the following definition of *CSuffix*:

type *CSuffix* τ α = *SSuffix* τ (*TimeVarying* α)

The use of segmented suffixes allows us to easily switch between different segments of time-varying values, thus avoiding the problem described in Subsection 4.1.2.

A type *TimeVarying* α can be simply defined as *Time* $\to \alpha$. Elliott gives a slightly more elaborate implementation that allows constant values to be handled specially. Our own idea is to represent values of *TimeVarying* as I/O actions that, when called, yield the respective current value. The advantage of this approach is that continuous suffixes can also mirror external sources of changing values.

However, care must be taken that sampling a time-varying value multiple times during the same event handling cycle does not lead to conflicting results. Finalizing our continuous suffix implementation is a goal for the future.

Often, we have continuous suffixes that represent physical phenomena like location or velocity of an object. Such suffixes can typically be defined by systems of recursive equations. With FRP systems like Fran and Yampa, such equation systems can be directly translated into Haskell code. However, our approch to implementing continuous suffixes does not allow for recursive definitions of suffixes. However, the solutions of recursive equation systems can often be expressed as convolutions. We want to explore how support for suffix convolution can be implemented, and whether such a feature can make recursive suffix definitions obsolete.

Segmented suffixes can be inefficient if their values are large data structures. For example, the contents of a list view in a GUI could be specified by a segmented suffix of lists. However, if the suffix would be updated, the list view would just receive a new list. So it would replace its contents completely. Furthermore, if a combinator would be applied to suffixes of lists, an update of one of the source suffixes would trigger a complete computation of the new value of the result suffix. A solution is to provide specific support for segmented suffixes of collections where updates are done incrementally. FranTk contains an implementation of this idea [26, Section 7.5], although we suppose that it does not meet the time bounds it is expected to comply with. We have already started to improve on the techniques used in FranTk and want to further develop this work in the future.

Last, but not least, we have discovered that concepts from temporal logic correspond to concepts from FRP by means of a Curry–Howard correspondence. As a result, ideas from temporal logic can be taken over to FRP, leading to improvements in FRP's interface and implementation. Pursuing this route further is an important research goal of us.

9.2. Further Work on Records

An open question is whether there are any performance issues involved in our implementation of records. After all, the linked-list implementation makes already simple field selection take linear time. Record conversion takes quadratic time since it contains two nested inductions. However, when record combinators are finally used in application code, their types are usually statically known. So it should be possible in principle to shift iteration over record schemes to compile time by using massive inlining. We still have to investigate whether GHC can do such inlining for us.

While using inductively defined record combinators is easy, implementing them is not, because values need to be wrapped and unwrapped. However, the task of writing all the necessary boilerplate code is rather mechanical. So it is likely that the boilerplate code can be generated by using Template Haskell [30], for example.

A similar problem with verbose code occurs when emulating subkinds. So it

might still be a good idea to provide subkind support as a language extension. Our record system would also profit from language support for names that would free us from explicitly declaring name types. However, keep in mind that our technique for pattern matching relies on names being represented by data constructors at the value level. Language-based name support should take this into account.

Existing proposals for record language support do not cover record type families, record scheme induction, and support for sorts of arbitrary subkinds. Since we have found these features to be very useful in practice, we argue that language support for records should not disallow them. It is probably best if the language provides only some basic support for record systems, and full record systems are then build on top of this as libraries.

A. Use of the Author's Work on Records

Chapters 7, 8, and 9 contain substantial portions of an earlier publication of the author [12], whose copyright is owned by the ACM. According to Subsection 2.5 of version 4 of the ACM copyright policy[1], ACM gives permission to use these portions here, but requires to quote the copyright notice of the original publication. This notice is as follows:

> Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
>
> PPDP '10 July 26–28, 2010, Hagenberg, Austria
> Copyright © 2010 ACM 978-1-4503-0132-9/10/07. . . $10.00

[1] See `http://www.acm.org/publications/policies/copyright-policy-v4#Retained`.

Bibliography

[1] F. Warren Burton. Indeterminate behavior with determinate semantics in parallel programs. In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 340–346, New York, 1989. ACM.

[2] F. Warren Burton. Encapsulating non-determinacy in an abstract data type with determinate semantics. *Journal of Functional Programming*, 1(1):3–20, January 1991.

[3] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *Proceedings of the 32nd ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages (POPL '05)*, pages 1–13, New York, 2005. ACM.

[4] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In Peter Sestoft, editor, *Programming Languages and Systems*, volume 3924 of *Lecture Notes in Computer Science*, pages 294–308. Springer, Berlin/Heidelberg, Germany, 2006.

[5] Antony Courtney. Frappé: Functional reactive programming in Java. In I. V. Ramakrishnan, editor, *Practical Aspects of Declarative Languages*, volume 1990 of *Lecture Notes in Computer Science*, pages 29–44. Springer, Berlin/Heidelberg, Germany, 2001.

[6] Conal Elliott. Functional implementations of continuous modeled animation (expanded version). Technical Report MSR-TR-98-25, Microsoft Research, Redmond, Washington, July 1998.

[7] Conal Elliott. Push-pull functional reactive programming. In *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell (Haskell '09)*, pages 25–36, New York, 2009. ACM.

[8] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, pages 263–273, New York, 1997. ACM.

[9] Benedict R. Gaster and Mark P. Jones. A polymorphic type system for extensible records and variants. Technical Report NOTTCS-TR-96-3, University of Nottingham, Nottingham, England, November 1996.

[10] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. In Johan Jeuring and Simon Peyton Jones, editors, *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer, Berlin/Heidelberg, Germany, 2003.

[11] Wolfgang Jeltsch. Improving push-based FRP. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Draft Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP '08)*, pages 179–193, Nijmegen, The Netherlands, May 2008. Radboud Universiteit Nijmegen.

[12] Wolfgang Jeltsch. Generic record combinators with static type checking. In *Proceedings of the 12th International ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP '10)*, pages 143–154, New York, 2010. ACM.

[13] Wolfgang Jeltsch. Signals, not generators! In Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, editors, *Proceedings of the Tenth Symposium on Trends in Functional Programming (TFP '09)*, volume 10 of *Trends in Functional Programming*, chapter 10, pages 145–160. Intellect, Bristol, England, 2011.

[14] Mark P. Jones and Simon Peyton Jones. Lightweight extensible records for Haskell. In Erik Meijer, editor, *Proceedings of the 1999 Haskell Workshop*, pages 55–66, Utrecht, The Netherlands, 1999. Universiteit Utrecht.

[15] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. Technical Report SEN-E0420, Centrum Wiskunde & Informatica, Amsterdam, The Netherlands, August 2004.

[16] Jon Pérez Laraudogoitia. Supertasks. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Stanford, California, spring 2011 edition, March 2011. `http://plato.stanford.edu/archives/spr2011/entries/spacetime-supertasks/`.

[17] John Launchbury and Simon Peyton Jones. State in Haskell. *LISP and Symbolic Computation*, 8(4):293–341, December 1995.

[18] Daan Leijen. Extensible records with scoped labels. In Marko van Eekelen, editor, *Proceedings of the Sixth Symposium on Trends in Functional Programming (TFP '05)*, volume 6 of *Trends in Functional Programming*, chapter 12, pages 179–194. Intellect, Bristol, England, 2007.

[19] Hai Liu and Paul Hudak. Plugging a space leak with an arrow. *Electronic Notes in Theoretical Computer Science*, 193:29–45, November 2007.

[20] Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1):1–13, January 2008.

[21] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '09)*, pages 1–20, New York, 2009. ACM.

[22] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*, pages 51–64, New York, 2002. ACM.

[23] Simon Peyton Jones. Tackling the awkward squad: Monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. In Tony Hoare, Manfred Broy, and Ralf Steinbrüggen, editors, *Engineering Theories of Software Construction*, volume 180 of *NATO Science Series, III: Computer and Systems Sciences*, pages 47–96. IOS Press, Amsterdam, The Netherlands, 2001.

[24] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.

[25] Meurig Sage. FranTk – a declarative GUI language for Haskell. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00)*, pages 106–117, New York, 2000. ACM.

[26] Meurig Sage. *Declarative Support for Prototyping Interactive Systems*. PhD thesis, University of Glasgow, Glasgow, Scotland, March 2001.

[27] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. In *Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP '08)*, pages 51–62, New York, 2008. ACM.

[28] Neil Sculthorpe and Henrik Nilsson. Optimisation of dynamic, hybrid signal function networks. In Peter Achten, Pieter Koopman, and Marco T. Morazán, editors, *Proceedings of the Ninth Symposium on Trends in Functional Programming (TFP '08)*, volume 9 of *Trends in Functional Programming*, chapter 7, pages 97–112. Intellect, Bristol, England, 2009.

[29] Neil Sculthorpe and Henrik Nilsson. Safe functional reactive programming through dependent types. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming (ICFP '09)*, pages 23–34, New York, 2009. ACM.

[30] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell (Haskell '02)*, pages 1–16, New York, 2002. ACM.

[31] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP '06)*, pages 251–262, New York, 2006. ACM.

[32] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation (PLDI '00)*, pages 242–252, New York, 2000. ACM.